Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

2008-03

A framework for software reuse in safety-critical system of systems

Warren, Bradley R.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/4142

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A FRAMEWORK FOR SOFTWARE REUSE IN SAFETY-CRITICAL SYSTEM OF SYSTEMS

by

Bradley Reed Warren

March 2008

Thesis Advisor:	James B. Michael
Co-Advisor:	Man-Tak Shing

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2008	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE A Framework for Software Reuse in Safety-Critical System of Systems		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Bradley Reed Warren			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

This thesis concerns the effective and safe software reuse in safety-critical system-of-systems. Software reuse offers many unutilized benefits such as achieving rapid system development, saving resources and time, and keeping up technologically in an increasingly advancing global environment. System software needs to be designed for both reuse and safety and available information shared effectively. We introduce a process neutral framework for software reuse in safety-critical system of systems. That framework consists of four elements: organizational factors, component attributes, component specification, and safety analysis. We developed a model (C^5RA) to capture the relevant component information and assist in specification matching. We conducted a survey of software safety metrics, created metrics, and developed a ranking. We applied the framework utilizing the reuse of a generic avionics software component. Our key findings are that congruence between all elements is required; software should posses certain attributes with metrics that support a safe design; software component information can be specified using C^5RA; and a process was identified for a system-of-systems hazard analysis for software reuse. The framework outlined provides a solution that enables effective software reuse in safety-critical system of systems.

14. SUBJECT TERMS Software reuse, system of systems, safety critical, software certification, safety, hazard analysis.			**15. NUMBER OF PAGES** 99
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

A FRAMEWORK FOR SOFTWARE REUSE IN SAFETY-CRITICAL SYSTEM OF SYSTEMS

Bradley R. Warren
Major, Australian Army
B.E., University of New South Wales, Australia, 1996
MBA., La Trobe University, Australia, 2006

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2008

Author: Bradley R. Warren

Approved by: Professor James B. Michael
Thesis Advisor

Professor Mantak Shing
Co-Advisor

Professor Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis concerns the effective and safe software reuse in safety-critical system-of-systems. Software reuse offers many unutilized benefits such as achieving rapid system development, saving resources and time, and keeping up technologically in an increasingly advancing global environment. System software needs to be designed for both reuse and safety, and available information needs to be shared effectively. We introduce a process neutral framework for software reuse in safety-critical system of systems. That framework consists of four elements: organizational factors, component attributes, component specification, and safety analysis. We developed a model (C^5RA) to capture the relevant component information and assist in specification matching. We conducted a survey of software safety metrics, created metrics, and developed a ranking. We then applied the framework utilizing the reuse of a generic avionics software component. Our key findings are that congruence between all elements is required; software should posses certain attributes with metrics that support a safe design; software component information can be specified using C^5RA; and a process identified a system-of-systems hazard analysis for software reuse. The framework outlined provides a solution that enables effective software reuse in safety-critical system of systems.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1
A. THE EVOLUTION OF SOFTWARE REUSE....1
B. CONCEPTS AND DEFINITIONS....4
1. Software Reuse....4
2. System Safety....6
3. Software Safety....9
4. Safety Critical Software....10
5. Safety Critical Environment....12
6. System of Systems....12
7. Safety-Critical System of Systems....13
8. Framework for Software Reuse....13

II. THE FEDERAL AVIATION ADMINISTRATION APPROACH TO SOFTWARE REUSE....15
A. DO-178B SOFTWARE CONSIDERATIONS IN AIRBORNE SYSTEMS AND EQUIPMENT CERTIFICATION....15
1. Overview....15
B. FAA AC20-148 REUSABLE SOFTWARE COMPONENTS (RSC)....16
1. Overview....16

III. FRAMEWORK FOR SOFTWARE REUSE IN SAFETY-CRITICAL SYSTEM OF SYSTEMS....23
A. OVERVIEW....23
B. ORGANIZATIONAL FACTORS....25
1. Culture....25
2. People....26
3. Structure....28
4. Reuse Domain....29
5. Reuse Potential....29
6. Reuse Capability....29
7. Policies, Processes and Practices....30
8. Reuse Metrics....33
9. Conclusion....34
C. REUSABLE SOFTWARE COMPONENT ATTRIBUTES....34
1. Overview....34
2. Definition of Software Component....35
3. Component Attributes....35
D. REUSABLE SOFTWARE COMPONENT SPECIFICATION....41
1. Overview....41
2. Software Component Specification....41
E. SAFETY PROCESS AND HAZARD ANALYSIS....46
1. Overview....46
F. METRICS....51

1. Overview 51
2. Software Safety Metrics 51
3. Summary of Software Safety Metrics 62
G. REGULATOR NEEDS 64
H. SUMMARY 64
IV. APPLICATION OF THE FRAMEWORK 65
A. APPLICATION OF THE FRAMEWORK 65
1. Example Process Applying the Framework 65
B. CONCLUSION 68
V. CONCLUSION 69
A. KEY FINDINGS AND ACCOMPLISHMENTS 69
B. FUTURE WORK 70
APPENDIX: EXAMPLE SOFTWARE LEVELS OF RIGOR 73
A. EXAMPLE SOFTWARE LEVEL OF RIGOR (LOR) MATRIX AND REQUIRED LEVEL OF RIGOR SOFTWARE PRODUCTS 73
B. SOFTWARE RISK ASSESSMENT MATRIX AND SOFTWARE HAZARD RISK INDEX 75
LIST OF REFERENCES 77
INITIAL DISTRIBUTION LIST 81

LIST OF FIGURES

Figure 1. AS/NZS 4360:2004 Risk Management Process Overview 8
Figure 2. MIL-STD-882D System Safety Process.....9
Figure 3. Framework for Software Reuse in Safety-Critical System of Systems 24
Figure 4. Example Process Showing Application of the Framework 65

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. 178B Software Levels....17
Table 2. DO-178B Deliverables....18
Table 3. Software Safety Metrics Ranking and Relevance....64
Table 4. Example Software Level of Rigor Matrix....73
Table 5. Example Required Level of Rigor Software Products....74
Table 6. Software Risk Assessment Matrix....75
Table 7. Software Hazard Risk Index....75

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF EQUATIONS

Equation 1: Desired Value of SRUTC Over Time 54
Equation 2. Software Requirements Demonstration Metric 54
Equation 3. Percent Software Safety Requirements 55
Equation 4. Percent Software Hazards 56
Equation 5. Controls with Causes 56
Equation 6. Verifications with Controls 57

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to express my appreciation to the Australian Army for providing me with the opportunity to pursue further professional development through a Master of Science in Software Engineering at the Naval Postgraduate School.

I would like to express my appreciation to my advisors, Professor Michael and Professor Shing, for inspiring my research in this area and for their guidance through my research. I would also like to thank them for their instruction in software engineering subject areas and for providing me with the foundations for my future software engineering career. I would also like to thank the other faculty members in the software engineering department for their support during my study.

I would like to thank my wife, Emily, for her dedicated support over the duration of my research and for her patience during the long nights and weeks of study and research. To her I owe an enormous debt and am eternally thankful that she was there for me over this time.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. THE EVOLUTION OF SOFTWARE REUSE

Software reuse evolved from rudimentary techniques, such as development of function calls in early programming languages (e.g., FORTRAN) and libraries of software routines for performing scientific calculations, to modern-day approaches to reuse that cover the entire software lifecycle and all software system artifacts. For instance, there are now well established methods for reusing software architectures and design patterns. The inclusion of high-level artifacts such as use cases and requirements has been in part due to the realization that decisions made early in the development process typically have a significant impact on software engineers' ability to refine the these artifacts into a machine interpretable or executable system. Software reuse relies on preplanning to enact a sustainable reuse program that meets the needs of the organizations involved in a system development project. Preplanning distinguishes software reuse from software salvage, an unsystematic, opportunistic approach to using software system artifacts not developed with reuse in mind.

Another driver for continued innovation in software reuse is what has been referred to as the "software crisis"[1]: the imbalance between the explosive growth in software demand and both lagging software development productivity and the shortage in the supply of software professionals. This "software crisis" is just as *apropos* today as it has been over the past forty years of software engineering. The success rate of delivering or maintaining information systems on schedule, at cost, and within performance constraints has not improved since the 1970s, as evidenced by the results of studies reported by the U.S. Government Accountability Office and the Standish Group. When applied in an appropriate manner, software reuse provides a means for improving software quality, increasing development productivity, shortening time-to-market (i.e., creating competitive advantage), achieving consistent application functionality, reducing

1 NATO Software Engineering Conference, NATO Science Committee, Garmisch, Germany, 7-11 October 1968.

risk of cost and schedule overruns, improving validation of user requirements through prototyping, leveraging of technical skills and knowledge, and avoiding the inclination to "reinvent the wheel" (e.g., a survey performed in 1983 found that 85% of software developed was not unique[2]).

In 2003, U.S. Department of the Navy adopted an open architecture initiative[3] to realize some of the aforementioned improvements in software development. The Navy's open architecture is a multi-faceted strategy providing a framework for developing joint interoperable systems that adapt and exploit open-system design principles. This framework includes a set of principles, processes, and best practices that: provide more opportunities for competition and innovation; rapidly field affordable, interoperable systems; minimize total ownership cost; optimize total system performance; yield systems that are easily developed and upgradeable; and achieve component software reuse. Thus, there is a recognized need to achieve greater software component reuse, which will subsequently support the achievement of other outcomes of the Navy initiative. There is also the need to avoid software duplication, save resources, and get more out of existing resources to keep Navy and the other U.S. services positioned as the best in the world at an acceptable cost.

Without software reuse, it is unlikely that an organization will achieve a technological advantage or be at the cutting edge of technology development because it will simply take too long or be too costly to achieve that position via either green-fields development (i.e., starting from scratch) or software salvage.

Although the rationale for software reuse is sound, there are significant challenges to applying reuse in the development of safety-critical applications in a system of systems. A system of systems is an amalgamation of legacy systems and developing systems that provide an enhanced capability greater than that of any of the individual

2 E.J. Joyce., Reusable Software: Passage to Productivity. Datamation, Volume 34, Number 18, Spetember 15, 1988, p. 98.

3 Department of Defense Directive 5000.1, The Defense Acquisition System, 12 May 2003.

systems within the system of systems.[4] System of systems are a great departure from standalone systems. There is uncertainty and risk associated with assumptions about the interfaces between the component systems and issues of system interoperability. Also certain rules and restrictions apply to the reuse of software in safety critical systems. Planning for reuse and determining the appropriateness of reuse in a safety-critical system of systems represent a significant challenge because of the large number of potential system configurations, their non-stationary position, the associated emergent hazards, and derived safety requirements. Furthermore, safety is a system property and thus reused software will impact on the system safety.

There are also challenges associated with using non developmental items (NDIs) in a system of systems. The challenges arise because a single stakeholder does not control these individual systems and these systems may be used in plug-and-play arrangements, leading to multiple possible configurations, which will result in emergent safety properties and requirements.

Overcoming the reusability challenges means that more software will be reused, software development productivity will increase and the quality of the software and system will improve. Even after applying these rules you do not necessarily have a safe design, because even the best design cannot fully isolate the safety critical functionality from the reused component or non-developmental item. This currently limits the extent that software can be reused in safety-critical systems, including that in a system of systems. A software component that possesses certain essential attributes that are explicitly revealed and is classified appropriately will be more readily available for use and reuse in a safety-critical system of systems. A framework describing these software component requirements will facilitate effective reuse in safety-critical system of systems, creating advantages for both the U.S. and Australian defense communities in developing safety-critical systems. Many military systems today are both safety critical and composed of legacy and new development systems, such as the U.S. Ballistic Missile Defense System.

4 D.S. Caffall., J.B. Michael., Architectural Framework for a System-of-Systems, in Proc of the IEEE International Conference on Systems, Man and Cybernetics, Volume 2, 10-12 Oct 2005, p1876.

B. CONCEPTS AND DEFINITIONS

1. Software Reuse

Software reuse is the use of existing software artifacts in the development of other software artifacts with the goal of improving productivity and quality, among other factors.[5] Software reuse can also be described as the process of leveraging components from one system or environment for use in other systems or environments with no or minimal change to the component. Environment in this case refers to the context of use of the system, such as the classic example of reusing the guidance and control software from Arianne 4 in Arianne 5 or the case of software evolution, where the requirements and context of a system changes with changes in its mission. Software reuse differs from software salvage in that it occurs in a systematic way with a degree of preplanning for future use. Software reuse results from effective foresight during development, providing an investment in that future use. Conversely, software salvage is an unsystematic opportunistic approach to using software-system artifacts that were not developed with reuse in mind. Software artifacts are the products or byproducts of the software development process that comprise possible candidates for reuse. Lim identifies ten types of software artifacts that may be reused. They are:

1. Architectures
2. Source Code
3. Data
4. Designs
5. Documentation
6. Estimates
7. Human Interfaces
8. Plans
9. Requirements
10. Test Cases

According to Lim, a software component is a set of software artifacts that comprise a coherent module. It may contain many of the artifacts described above.[6] There are, however, many different definitions for a software component. Braude defines a software component as a software collection used without alteration, under the goal of

5 W.C. Lim., Managing Software Reuse, A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components, Upper Saddle River, NJ.: Prentice Hall PTR, 1998, p7.

6 W.C. Lim., Managing Software Reuse, A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components, Upper Saddle River, NJ.: Prentice Hall PTR, 1998, p539.

reuse and as the vehicle for delivering reuse.[7] Alternatively, Larman states that a component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.[8] Pressman, on the other hand, defines a software component as a unit of composition with contractually specified and explicit context dependencies only.[9] Although there may be many different definitions, there is much commonality among them. In this thesis we define a software component as a collection of software comprising a module with a well-defined purpose that may be used with no or minimal alteration.

Lim identifies the many benefits of software reuse as:

- Improved software quality
- Increased development productivity
- Shortened time to market
- Consistent application functionality
- Reduced risk of cost and schedule overruns
- Allows prototyping for validating user requirements
- Leveraging of technical skills and knowledge[10]

Software reuse, once established, enables an organization to accomplish more without additional resources. This is important for both businesses trying to gain and maintain a competitive advantage and public sector entities attempting to acquire the best systems at the least cost. Software reuse allows an organization to generate competitive advantage and to avoid duplicating past efforts. Furthermore, it is one method for mitigating the software crisis. Software reuse is a means for an organization to leverage

7 E. Braude., Software Design: From Programming to Architecture, Hoboken, NJ.: John Wiley & Sons, 2004, p385.

8 C.Larman., Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition, Upper Saddle River, NJ.: Pearson Education, 2005, p654.

9 R.S. Pressman., Software Engineering A Practitioner's Approach, Sixth Edition, New York, NY.: McGraw-Hill, 2005, p817.

10 W.C. Lim., Managing Software Reuse, A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components, Upper Saddle River, NJ.: Prentice Hall PTR, 1998, p102.

past successes through organizational learning and institutional memory. Software reuse represents a strategy for meeting the challenges of a rapidly changing complex environment.

These benefits, however, may not be realized if reuse is not implemented effectively. For example, reusing a software component that subsequently does not meet the requirements, which may not be evident until verification and validation activities, may involve increased work and costs because new software may be developed later than originally intended. Without organizational support for reuse, including incentives for reuse, these benefits may be overlooked and go largely unrealizable.

2. System Safety

Safety is defined as freedom from those conditions that can cause death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment.[11]

In other words, safety is freedom from conditions that cause accidents (mishaps). Furthermore, safety is the execution of a software product in a system without causing the system to exist in a hazardous state.[12]

MIL-STD-882Ddefines system safety as:

> the application of engineering and management principles, criteria, and techniques to achieve acceptable mishap risk, within the constraints of operational effectiveness and suitability, time, and cost, throughout all phases of the system life cycle.[13]

Furthermore, MIL-STD-882D defines system safety engineering as:

> An engineering discipline that employs specialized professional knowledge and skills in applying scientific and engineering principles,

11 Department of Defense, Standard Practice for System Safety, MIL-STD-882D, 10 February 2000.

12 R. Singh., A Systematic Approach to Software Safety, in Proc of the sixth Asia Pacific Software Engineering Conference (APSEC '99), 1999.

13 Department of Defense, Standard Practice for System Safety, MIL-STD-882D, 10 February 2000.

> criteria, and techniques to identify and eliminate hazards, in order to reduce the associated mishap risk.[14]

In these cases MIL-STD-882D defines a hazard as:

> Any real or potential condition that can cause injury, illness, or death to personnel; damage to or loss of a system, equipment or property; or damage to the environment.[15]

A hazard can also be defined as a source of potential harm[16] or a prerequisite to a mishap or accident.

The system safety process is a specific application of the risk management process to the system safety domain where instead of risk, the terms hazard and mishap risk are used. The objective of system safety is to achieve an acceptable mishap risk through a systematic approach of hazard analysis, risk assessment, and risk management.[17] Figure 1 illustrates the risk management process as defined in the Australian and New Zealand standard AS 4360.[18]

14 Department of Defense, Standard Practice for System Safety, MIL-STD-882D, 10 February 2000.

15 Department of Defense, Standard Practice for System Safety, MIL-STD-882D, 10 February 2000, p2.

16 AS/NZS 4360:2004. Australian / New Zealand Standard. Risk Management, p3.

17 Department of Defense, Standard Practice for System Safety, MIL-STD-882D, 10 February 2000, p3.

18 AS/NZS 4360:2004. Australian / New Zealand Standard. Risk Management.

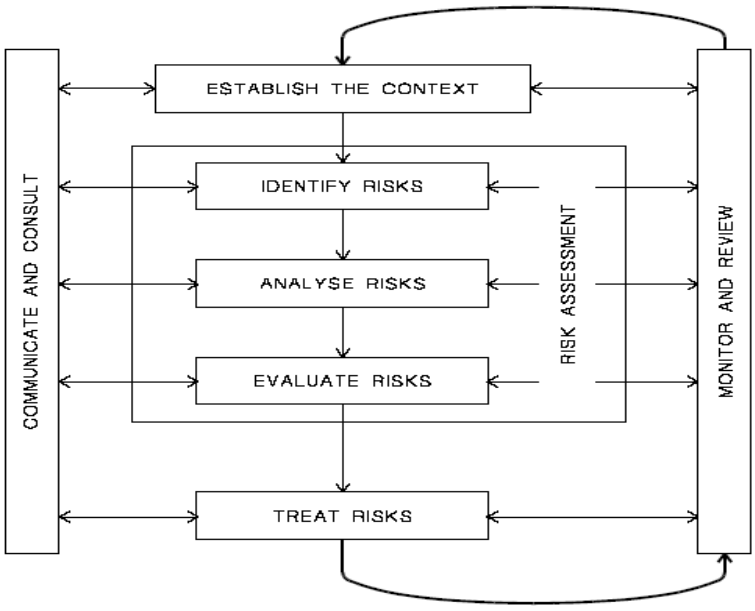

Figure 1. AS/NZS 4360:2004 Risk Management Process Overview

The system safety process as defined by MIL-STD-882D is as follows:

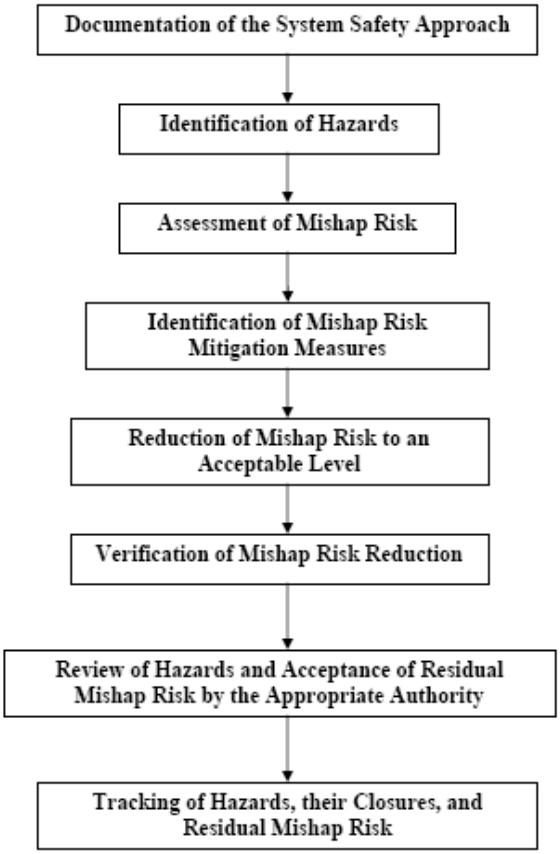

Figure 2. MIL-STD-882D System Safety Process

Both processes are similar, involving a number of common steps. Neither process has a way of guaranteeing that all relevant risks or hazards will be identified, which is why sub-processes are required to ensure that those relevant risks or hazards are identified so they can be treated.

3. Software Safety

The discipline of software safety engineering is the systematic approach to identifying, analyzing, and tracking software mitigation and control of hazards and hazardous functions (e.g., data and commands) to ensure safer software operation within a system.[19]

19 Overview of Software Safety, NASA Software Safety, May 2007, http://sw-assurance.gsfc.nasa.gov/disciplines/safety/index.php.

Software system safety is a subset of system safety and, according to Leveson, "implies that the software will execute within a system context without contributing to hazards."[20] Software safety is a discipline within system safety engineering that focuses on the system's software and its interactions with the hardware and human operators. Software itself cannot cause harm, except possibly emotional harm or stress (nerves) to those involved in its development; however, through its interactions with hardware it can cause hazards that may result in mishaps. Software safety is an engineering and management approach to ensuring that the software minimizes the system safety risk to an acceptable level while maintaining the effectiveness of the software and system. It is focused on meeting safety requirements. The purpose of system safety and software safety is to identify and mitigate hazards associated with the operation and maintenance of the system and software respectively and to define the residual risk, based on success of the controls implemented.[21]

Software safety is concerned with both removing defects in software artifacts for which there would be hazardous consequences and ensuring the system's specification adequately captures the safety requirements. Software safety is an ongoing process as the software and operational environment evolves.

4. Safety Critical Software

Safety critical is a term applied to any condition, event, operation, process, or item whose proper recognition, control, performance, or tolerance is essential to safe system operation and support (e.g., safety critical function, safety critical path, or safety critical component).[22]

---

20 N. Leveson., Safeware: System Safety and Computers. Addison Wesley, Boston, 1995.

21 V. Basili., K. Dangle., L. Esker., F. Marotta., Gaining Early Insight into Software Safety: Measures of Potential Problems and Risks, in Proc of the Systems & Software Technology Conference, June 2007, p3.

22 Department of Defense, Standard Practice for System Safety, MIL-STD-882D, 10 February 2000.

According to NASA,[23] software is safety critical if it meets at least one of the following criteria:

1. Resides in a safety-critical system (as determined by a hazard analysis) and at least one of the following:

    a. Causes or contributes to a hazard

    b. Provides control or mitigation for hazards

    c. Controls safety-critical functions

    d. Processes safety-critical commands or data

    e. Detects and reports, or takes corrective action, if the system reaches a specific hazardous state

    f. Mitigates damage if a hazard occurs

    g. Resides on the same system (processor) as safety-critical software

2. Processes data or analyzes trends that lead directly to safety decisions (e.g., determining when to turn power off to wind tunnel to prevent system destruction)

3. Provides full or partial verification or validation of safety-critical systems, including hardware or software subsystems.

A safety critical function is any function, whether hardware, software, operator, or combination thereof, that directly influences a hazard or hazardous situation. Furthermore, it is any function whose improper functioning could result in a hazard in which improper functioning includes failure modes, out of tolerance conditions, timing error or problems (e.g., data latency), or other errors.[24]

---

23 Overview of Software Safety, NASA Software Safety, May 2007, http://sw-assurance.gsfc.nasa.gov/disciplines/safety/index.php.

24 W.C. Lim., Managing Software Reuse, A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components, Upper Saddle River, NJ.: Prentice Hall PTR, 1998.

In the context of safety critical software, there is the rule that reused software or non-developmental items (NDI), be it COTS, GOTS, MOTS[25] or legacy software, may not initiate, sustain, or prevent occurrence of a safety critical function unless the NDI has been evaluated to the same level as a developmental software component. This rule results from the reality that not enough information is typically provided on reusable software components and that there is not a standard or shared understanding to describe what that should be. Furthermore, safety is a system property, and thus reused software will impact system safety; it is dependent on the interfaces and interactions with the system. Thus, this suggests that there will always be a requirement for analysis of the component in its intended environment. This is likely to involve operational testing and is dependent on the safety requirements.

5. Safety Critical Environment

A safety critical environment is an environment in which there are potential hazards or hazardous situations or where proper system functioning is essential for safe operation.

6. System of Systems

A system of systems is an amalgamation of legacy systems and developing systems that provides an enhanced capability greater than that of any of the individual systems within the system of systems.[26] It represents a natural evolution from systems and systems development to now integrating several systems to form a greater system for the accomplishment of specific objectives.

System of systems represent a significant departure from standalone systems as their components are individual systems often with a high level of autonomy whereas system components, subsystems are often unable to operate in isolation from the other subsystems that compose the system. Furthermore, system of systems development is not

25 Commercial off the Shelf (COTS), Government off the Shelf (GOTS), and Modified off the Shelf (MOTS).

26 D.S. Caffall., J.B. Michael., Architectural Framework for a System-of-Systems, in Proc of the IEEE International Conference on Systems, Man and Cybernetics, Volume 2, 10-12 Oct 2005, p1876.

just a larger version of systems development; it represents an area of greater uncertainty and risk associated with assumptions and unknowns of the interfaces between the component systems. The component systems are often developed to meet requirements and constraints that are different from those of the system of systems. It is often the case in system of systems development that legacy systems are integrated with new systems and thus not all system of systems components were designed to optimize the performance or dependability of the system of systems or to conform to required interface and interoperability specifications.

Example system of systems are the U.S. Ballistic Missile Defense System (BMDS) and the U.S. Army's Future Combat System (FCS). The U.S. Commercial and Military Aviation systems could be described as further system of systems examples.

7. Safety-Critical System of Systems

A safety-critical system of systems is a system of systems operating in an environment in which there are hazards or hazardous situations or where proper system functioning is essential for safe operation within a system of systems. The environment represents the context in which the system will be used and in the case of a system of systems, there may be many different contexts of use. These many contexts of use will have a significant affect on safety because safety is context dependent. Safety is a function of the context of use, against which hazards can be identified.

Commercial and military aviation could be described as a safety-critical system of systems. In the case of commercial and military aviation, a number of different systems operate cohesively for a common purpose where safety is of paramount importance.

8. Framework for Software Reuse

Successful software reuse requires a holistic, systematic, multi-disciplinary approach which is matched to the specific context of use. I intend to establish a framework within which software reuse can be enabled and supported and the probability of a successful outcome increased in a safety-critical system of systems context.

This framework requires the integration of a number of factors to be successful.

Much of the success of any reuse program can be attributed to the amount of information available on a potential reuse candidate, the ability to search and locate that information, and supportive organizational policies which encourage reuse. To a lesser degree, success can also be attributed to institutionalized memory based on past experiences employing software reuse that will contribute to future organizational policies on software reuse.

This framework will establish the organizational policy and culture requirements, the information required on a reusable software component and its possible organization, metrics that are of interest, requirements from a regulator or certification authority's perspective, and the hazard analysis that must occur to make software reuse a success in safety-critical system of systems.

This thesis will focus on compositional reuse rather than generative reuse, as described by Lim.[27] Generative reuse software is a tool for producing software artifacts that may be compositionally reused; however, the framework may provide value in that area too.

Chapter II will provide an overview of standard DO-178B, Software Considerations in Airborne Systems and Equipment Certification, which will provide the context for discussion of an Advisory Circular on Reusable Software Components (AC20-148) issued by the Federal Aviation Administration (FAA). The Advisory Circular on Reusable Software Components provides an example of how reuse is encouraged and a process established to make it more prevalent and ubiquitous in the U.S. aviation community.

Chapter III introduces the framework for software reuse in safety-critical system of systems, the metrics relevant to safety, reuse metrics, regulator needs, and their influence on elements of the framework. Chapter IV provides an example of how to apply the framework, and Chapter V draws some conclusions and discusses future work.

27 W.C. Lim., Managing Software Reuse, A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components, Upper Saddle River, NJ.: Prentice Hall PTR, 1998, p397.

II. THE FEDERAL AVIATION ADMINISTRATION APPROACH TO SOFTWARE REUSE

A. DO-178B SOFTWARE CONSIDERATIONS IN AIRBORNE SYSTEMS AND EQUIPMENT CERTIFICATION

1. Overview

RTCA/DO-178B[28] Software Considerations in Airborne Systems and Equipment Certification is a contemporary software assurance standard that is highly regarded within the aviation community. It is the preferred standard for software assurance for safety related airborne software in the Australian Defence Force (ADF) and the U.S. Federal Aviation Administration (FAA) and is used by many others developing or regulating airborne software. DO-178B provides guidelines on software production for airborne systems and equipment. The standard could be applied across application domains but some aspects are avionics specific.

DO-178B applies to the assurance and certification of all software requirements and not just those that are safety related. In DO-178B, two main entities are described, those being the Certification Authority and the Applicant. The Certification Authority is the organization or person granting approval on behalf of the country responsible for aircraft or engine certification[29] and is the organization that defines certification requirements, conducts reviews of compliance with safety requirements, and certifies compliance with the requirements. The Applicant is the person or organization seeking approval from the Certification Authority and therefore is the party responsible for providing the argument and evidence for certification. The FAA uses DO-178B when certifying airborne software, and it is thus the standard that developers, integrators, and users of airborne software must comply with to achieve certification from the FAA. DO-178B is also the preferred standard of the ADF for assurance of airborne software.

---

28 Radio Technical Commission for Aeronautics, DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification, Washington D.C., 1 December 1992.

29 Ibid., p. 12.

B. FAA AC20-148 REUSABLE SOFTWARE COMPONENTS (RSC)

1. Overview

According to Wlad, no formal guidance or standard exists that maximizes the utility of software reuse in any industry.[30] For the development of software for use in safety-critical system of systems, a the Federal Aviation Administration (FAA) issued a relatively new policy to provide some guidance on the certification requirements when an already certified software component is to be reused, but this does not focus on the design for reuse or those inherent attributes that the software component must exhibit to facilitate effective reuse. Wlad asserts that this new policy will cause a major shift in how software is reused in safety critical systems. The new policy is a Reusable Software Components Advisory Circular (AC20-148) published in 2004. This circular details the approach and documentation necessary for systematic reuse of software components that meet the guidelines of DO-178B. AC20-148 now allows for acceptance of software independent of a hardware platform, enabling developers to take certification credit on one project and apply it to future projects.

The FAA asserts that this approach is one acceptable means of compliance but not the only one to gain acceptance or credit for the reuse of a software component. As this is an advisory circular, it is not mandatory nor does it constitute a regulation. This AC only applies when all stakeholders agree that the software component is reusable because meeting the policy requirements is likely to require additional effort and resources. The AC may apply to verification and development tools, although the FAA plans to cover this specifically in future guidance. The motivation for this guidance comes from the economic incentives of reuse, encouraging software developers to develop reusable software components (RSCs) that can be integrated into many systems. DO-178B based verification of systems is known to be an expensive endeavor. Providing a mechanism for taking credit on one project and using it on another via use of an RSC reduces both

30 J. Wlad., Software Reuse in Safety critical Airborne Systems, in Proc of 25th Digital Avionics Systems Conference, Oct 15, 2006, p. 6C5-1.

certification cost, time, and risk. The guidance in the AC ensures that systems using RSCs meet all applicable RTCA/DO-178B objectives.

The FAA may grant acceptance of an RSC provided all stakeholders comply with the advisory circular and that no installation, safety, operational, functional, or performance concerns are identified. Essentially this AC increases the certification requirements and documentation for the component in the first instance, making it easier for subsequent uses of the software component. The AC makes a good point in that acceptance of the RSC for one project does not guarantee acceptance on a later project as applicants must consider installation, safety, operational, functional, and performance issues for each project. This is important because safety is a system property. Just because there are no safety issues in one environment does not preclude them from existing in the new environment, which is particularly important in a system of systems. In other words, the RSC is not to be reused blindly and should be assessed for each project or environment. The approach identified in the AC requires a close working relationship between the FAA, the developer, and other stakeholders in addition to the provision of evidence to support the argument that the RSC satisfies all DO-178B objectives throughout the development of the system. Table 1 shows the failure conditions of the safety requirements that have been allocated to software as the source and the number of objectives that require evidence. For example, a software failure condition that is catastrophic receives a level of A and requires evidence to support 66 objectives.

Failure Condition	Software Level	Number of Objectives
Catastrophic	Level A	66
Hazardous / Severe - Major	Level B	65
Major	Level C	57
Minor	Level D	28

Table 1. 178B Software Levels

The advisory circular recognizes that on a project the developer, integrator, and applicant for an RSC may be different, and even when an applicant plays all three roles, there needs to be a process for communicating and transferring accepted data among the relevant stakeholders. The advisory circular addresses that communication and transfer of data and identifies what information each party to the RSC must produce. The advisory circular recognizes the need for a greater level of DO-178B compliance and evidence thereof through documentation when applying for acceptance of an RSC. The advisory circular is also more specific than DO-178B as to what the DO-178B deliverables should contain to comply with the advisory circular. Table 2 lists the DO-178B deliverables required for certification.

Plan for Software Aspects of Certification
Software Development Plan
Software Verification Plan
Software Configuration Management Plan
Software Quality Assurance Plan
Software Standards (Reqs/Design/Code)
Software Requirements
Design description
Source code
Review results
Test procedures/Test results
Software Life Cycle Environment Index
Software Configuration Index
Problem reports
Software Configuration Management records
Software Quality Assurance records
Software Accomplishment Summary

Table 2. DO-178B Deliverables

The advisory circular also describes the data requirements that the developer must provide to the integrator or applicant. As an example, data requirements include failure conditions, safety features, protection mechanisms, architecture, limitations, software levels, interface specifications, and intended use of the RSC. Furthermore, open problem reports on the RSC and analysis of any potential functional, operational, performance, and safety effects must be provided; this is particularly important in assessing the appropriateness for a particular context of use and in transferring safety information. Another positive approach taken in the advisory circular is that regardless of any legal and proprietary issues and agreements about the delivery of software life cycle data (or artifacts) between the applicant and the developer, the data must be available for the certification authority or authorized designee's review and inspection. This places the interest of public safety above all other concerns. This means that complete information is disclosed to the certification authority even if the applicant or integrator does not have access to it, overcoming the potential adverse impact proprietary protection may place on safety.

The FAA also requires that the developer submit a data sheet for the RSC which must concisely summarize:

- RSC functions
- Limitations
- Analysis of potential interface safety concerns
- Assumptions
- Configuration
- Supporting data
- Open problem reports
- Software characteristics
- Other relevant information that supports the integrator's or applicant's use of the RSC

This data sheet also assists in increasing the understandability of the software component, characterizing what information is deemed important. There are also provisions and requirements placed on the developer for identifying and maintaining data to support changes to the RSC.

Military aviation regulators could take advantage of the approach described in the advisory circular when full information disclosure may not be achievable or required with all stakeholders but that information is made available to the regulator. Taking the approach identified in the AC may reduce some of the verification activities required on subsequent projects and reduce the requirement to redo DO-178B certification activities. The advisory circular also identifies potential software reuse issues and provides a process for recertification when changes are made to an RSC.

Overall the advisory circular provides thorough and systematic guidance for the acceptance and credit for reuse of a reusable software component. The advisory circular specifies the process, the information needs of all parties, some potential reuse problems, and how to gain recertification of an RSC after changes to it are made. The advisory circular recommends the use of DO-178B as the relevant standard for initial and future compliance; however, this is not mandatory but it would be more difficult if another standard was used. It recognizes the importance of a design for reuse approach and the allocation of appropriate resources to gain credit for the early work when reused. Just like safety and other issues, consideration of reuse issues as early as possible in the system development is much more efficient in assuring effective reuse than delaying consideration until after many of the design and development decisions have been made.

A deficiency of the advisory circular is that it does not specifically refer to the use of hazard analysis, which is a cornerstone of safety engineering, nor what the information requirements for providing evidence to support a safety case in the new context of use. A safety case sets out the safety justification for the system and contains a record of all the safety activities associated with a system throughout its life.[31] The advisory circular does

31 N. Storey., Safety-Critical Computer Systems, New York, NY.: Addison-Wesley, 1996, p364.

recommend that safety must be considered and addressed, which may be the authors' way of providing the evidence in support of the safety case for the component; however, this was not explicit.

As this advisory circular is not mandatory and represents guidance only, but refers to DO-178B as the certification basis, it must be recognized that this approach will be seen as the preferred and most common way. The advisory circular does not preclude the use of other ways of achieving RSC certification; however, it makes compliance with other standards more arduous by specifically referring to DO-178B, thereby perpetuating and encouraging its use. Thus this advisory circular becomes a de facto standard for reusable software component certification within U.S. civil aviation. The advisory circular is the FAA's recognition that reusable software components are important and a process to achieve certification and attain credit for subsequent uses is beneficial.

THIS PAGE INTENTIONALLY LEFT BLANK

III. FRAMEWORK FOR SOFTWARE REUSE IN SAFETY-CRITICAL SYSTEM OF SYSTEMS

A. OVERVIEW

According to Lim, software reuse is most effective when practiced systematically.[32] In order to achieve a systematic approach that is also holistic and multidisciplinary, a framework for software reuse in safety-critical system of systems is required for identifying the key aspects, factors, and influencers on software reuse and for generating discussion on them. It is therefore proposed that the successful application of software reuse in a systematic approach depends on the following four factors (or pillars):

1. Organizational Factors (Enabler or Foundation).
2. Reusable software component attributes (Quality).
3. Reusable software component specification (Knowledge Capture and Search Effectiveness).
4. Safety Analysis (Environmental Match).

This chapter will explore these four factors of software reuse in detail.

The particular context of safety-critical system of systems places greater emphasis, in particular, on factors 2 and 4 of the framework. What differentiates this framework from a framework for reuse in general is the level of detail in each element and that a safety or hazard analysis and in particular a system-of-systems hazard analysis is essential.

The framework depicted in figure 3 shows how the factors (or pillars) contribute to achieving effective software reuse in safety-critical system of systems. In this case, the framework refers to the supporting structure for software reuse. It is process neutral and represents the desired elements for successful software reuse.

32 W.C. Lim., Managing Software Reuse, A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components, Upper Saddle River, NJ.: Prentice Hall PTR, 1998, p. 10.

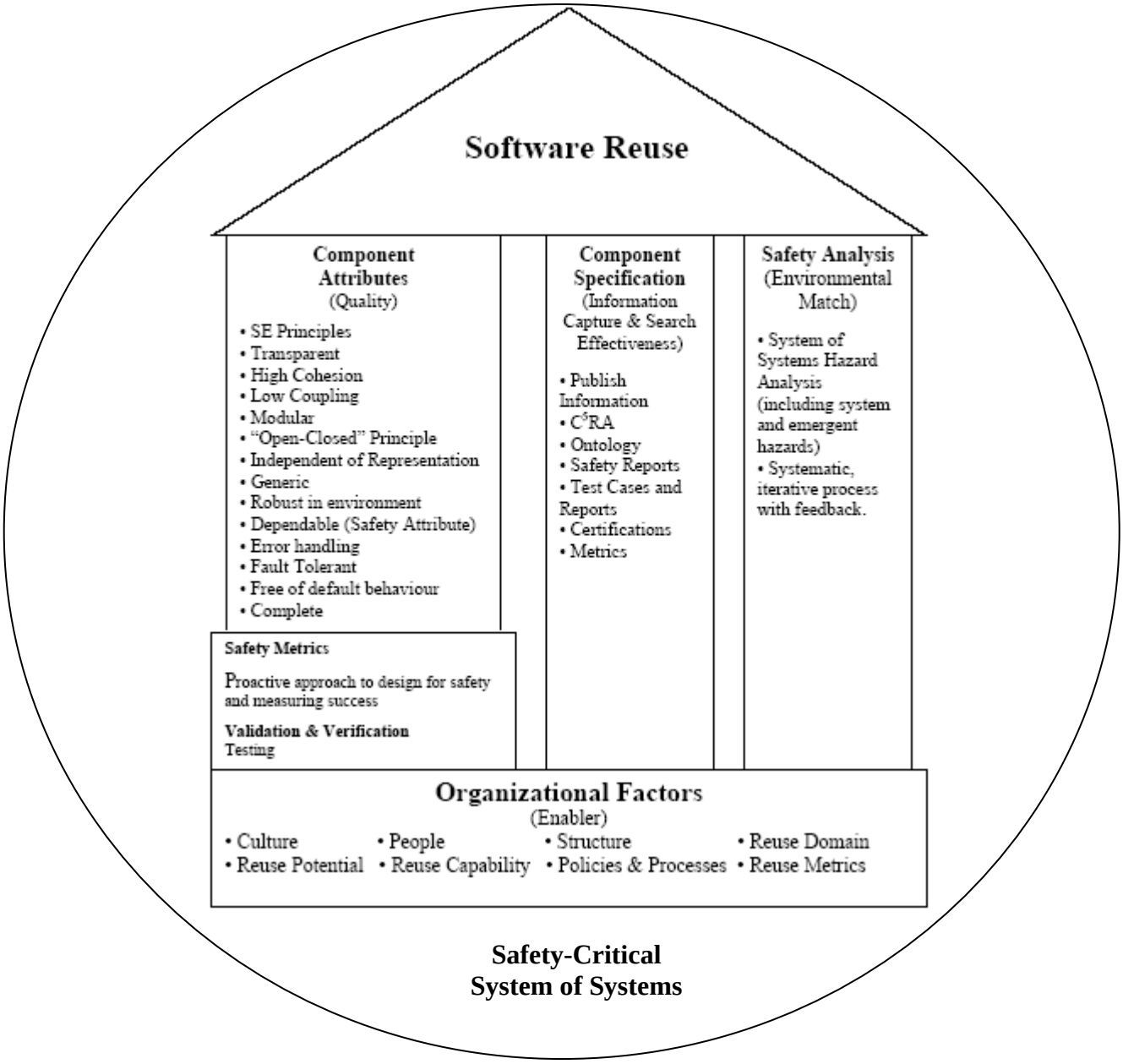

Figure 3. Framework for Software Reuse in Safety-Critical System of Systems

This framework focuses on compositional reuse, the construction of new software products by assembling existing reusable artifacts, rather than generative reuse. The framework, however, may provide value for the latter.

The needs and policies of the regulator will have an impact on any reuse program and will largely dictate the activities required and documentation necessary to obtain the benefits of software reuse. Example regulators include the FAA for civilian aviation in

the United States, the Naval Air Systems Command for U.S. Navy (USN) and U.S. Marine Corps (USMC) aviation, and the Australian Defence Force (ADF) for military aviation in Australia. The specific regulator for the safety-critical system of systems will have influence on the framework as evident in the discussion of the FAA's influence on software reuse through its guidance (Advisory Circular 20-148).

B. ORGANIZATIONAL FACTORS

Organizational factors represent the enabling and supporting functions within an organization that must exist to achieve successful software reuse in any context. These non-technical, management factors have significant influence on any software reuse activity and if not aligned appropriately, can dramatically affect the success of any proposed software reuse. Effective software reuse requires a strong commitment from senior management, a documented process that supports the organization's mission, a software reuse policy, and a delegated software reuse team. The significant factors affecting software reuse from an organizational perspective fall into the following categories: culture; people; structure; reuse domain; reuse potential; reuse capability; policies, processes and practices; and reuse metrics.

1. Culture

Organizational culture is defined as "consciously held notions shared by members that most directly influence their attitudes and behaviors."[33] These notions may include behavioral norms, values, beliefs, rituals, symbols, and behaviors that employees believe are expected of them to fit in and survive in an organization. Organizational culture can also be thought of as the leadership and management style, employee involvement and participation, and the customs and norms of an organization. These have a significant affect on the performance of an organization, which subsequently affects software reuse efforts. If those closely held notions within an organization are not conducive to software reuse, then it will be difficult for the organization to effectively reuse software. If the

33 R.H. Kilmann., M.J. Saxton., R. Serpa., Gaining Control of the Corporate Culture, Jossey Bass Business and management Series, San Francisco, CA.: 1985.

leadership and management style promotes reuse, this can aid in the successful implementation of a reuse program. A supportive organizational culture cannot guarantee the success of software reuse.. The lack of a supportive organizational culture, however, will almost certainly guarantee failure. The leadership and management of an organization have a significant influence in this regard, and they must both buy into software reuse and promote its benefits throughout the organization.

It is not easy to change an organizational culture, because this requires changing the ingrained ways of that which defines the organization, but change is necessary if software reuse is to be successful and implemented where required. Members of the organization with a stake in software development must buy in to the concept of software reuse and understand its purpose for it to be a success across the organization. For software reuse to be implemented successfully, it is essential that there be a culture of cooperation and collaboration between stakeholders of the system of systems to ensure all relevant issues are considered in a timely and effective manner. In the context of safety-critical system of systems what is desired is a safety culture that permeates all organizations that contribute to the system of systems. This reflects the ideal and may not be achievable in reality. A safety and reuse champion is required who will educate, influence, and align the other stakeholders to achieve the necessary organizational culture.

2. People

People are a key component in the implementation of software reuse in an organization. They must be motivated to reuse with the requisite support to enable them to do so. People must understand the purpose for reusing software and the way it is to occur. Incentives to encourage reuse must be used to affect attitudinal changes towards its practice. Without any one of these, the best efforts may come undone.

Furthermore, the right people need to be placed in the right positions to have a positive impact on the software reuse efforts of an organization. Possible positions could include:[34]

a. Reuse Champion,

b. Domain Analyst,

c. Domain Expert,

d. Domain Workproduct Manager,

e. Reuse Engineer,

f. Reuse Analyst,

g. Reuse Economist/Metrician,

h. Librarian, and

i. Reuse Manager.

If software reuse is to be successful, all the preceding roles should be played in large and in small organizations; however, you could have different numbers of people responsible for each position. For example, you may have one person for each position; one person fulfilling more than one position; or the responsibilities of each position being fulfilled by a team of people. It is essential for an effective reuse program to select the right staff for each position and educate, train, and motivate them.

A change in attitude and culture is important for long-term reuse success. The means for achieving a change in attitude and culture include education and training along with communication of the purpose of the reuse initiative. It is often the lack of communicating the goals of the reuse initiative that lets down the initiative and prevents it from being successful. Good management of the people filling the position cannot be

34 W.C. Lim., Managing Software Reuse, A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components, Upper Saddle River, NJ.: Prentice Hall PTR, 1998, pp225-227.

overstated in order to achieve success in software reuse. All the people responsible for software development and maintenance have a role in successfully implementing software reuse.

3. Structure

The typical organizational structures, such as functional, project, matrix, and combinations thereof, may be used for reuse. Each structure has its advantages and disadvantages; these are dependent on the context for reuse. The best structure to apply will depend on the characteristics of the organization, the reuse strategy, and the environment that it will occur in. It is important to choose the right organizational structure that will deliver the reuse strategy given the organizational characteristics and operating environment. In safety-critical system of systems in which component quality and adherence to safety requirements is critical for software reuse, the best organizational structure may be a functional organization, where there is consistency in personnel, greater retention of corporate knowledge (or institutional memory), and the ability to centralize reuse efforts and efficiently manage them. Furthermore, under a functional organization it is more likely that common standards and methods will be reinforced and applied to the particular reusable artifacts in each functional area. However, just as there is the potential for greater quality and consistency among particular reusable artifacts, the functional structure may not be as responsive to customer needs as the other organizational structures and suffer from coordination problems between the functional departments of the reusable artifacts thus creating issues when components are to be reused. An organization does not necessarily structure itself based on what is best for reuse, so that structure may not be suitable in all cases. In structuring an organization for reuse, it is also important to ensure that there are reuse departments responsible for the management of the reusable artifacts. These departments will maintain software repositories. An ecosystem supporting reused software in an organization is important.[35]

35 R. Anthony., Software Firms Reap Benefits of Code Reuse. The Wall Street Journal, December 3, 2007. www.livemint.com, http://www.livemint.com/2007/12/03233411/Software-firms-reap-benefits-o.html

4. Reuse Domain

The market segment the organization is serving, the types of products the organization produces, or the public sector the organization is in often define the reuse domain. This influences what form software reuse should take and how it is performed. Competition and regulation within the domain of reuse will shape software reuse, define what is involved, motivate its use, and impact on its ultimate success. A stable operating domain for the organization will assist in making reuse decisions and in exploiting reuse to its full potential. A domain characterized by frequent disruptive technologies may provide little impetus for establishing or investing in a software reuse program.

5. Reuse Potential

Lim refers to a reuse potential and aptitude model[36] to help firms identify their potential for reuse success. This model is based on the assumption that the organization must possess the requisite potential for reuse as well as the ability to successfully exploit that potential. Reuse potential represents the latent redundancies and opportunities within and across domains which, when combined with proper organizational ability, can become actual reuse.[37]

Latent competencies or opportunities must exist for the organization in order to have reuse potential. A number of organizational characteristics influence reuse potential and provide the supporting basis for future reuse.

6. Reuse Capability

In order to recognize reuse potential, an organization must possess the capability to exploit the potential. Reuse aptitude (capability) refers to the requisite ability or capacity to exploit the reuse potential.[38] This aptitude, capacity, or capability refers to the ability to turn potential into reality and consists of the organizational characteristics,

36 W.C. Lim., Managing Software Reuse, A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components, Upper Saddle River, NJ.: Prentice Hall PTR, 1998, pp 75.

37 Ibid., p. 75.

38 Ibid., p. 78.

competencies, resources, or access to resources in order to implement software reuse. Software reuse involves an investment and effort upfront, which can be up to forty percent more than software written without reuse in mind (not to mention the costs of establishing a reuse program initially).[39] Moreover, an organization requires those additional resources at the right time in order to implement reuse. If an organization does not have the resources or access to resources when required, it will not be able to successfully implement a reuse program and fulfill its reuse potential. The specific reuse capabilities required are referred to throughout this chapter.

7. Policies, Processes and Practices

The members of an organization must understand the purpose of software reuse and develop policies, processes, and practices that are supportive of reuse. The policies, processes, and practices must be supportive of software reuse for it to be a success. These policies and processes must be clearly defined and consistent with organizational culture and provide clear guidance on how it will be implemented and who is responsible, in much the same way as AC20-148 does. The policies, processes, and practices must incentivize software reuse, rewarding the practice and cementing it into the organizational culture. It is the policies, processes, and practices that will form an organization's culture over time, and to succeed reuse must become part of that culture. A reusability culture will be created over time through organizational policies, processes, and practices. Meyer defines reusability culture as one in which all software is developed under the assumption that it will be reused.[40] He also says that the reusability of any software should not be trusted until it has been reused. This essentially supports the treatment of reusability concerns throughout development and that just because software has been designed that way does not mean that it is reusable in another context without

39 R. Anthony., Software Firms Reap Benefits Of Code Reuse. The Wall Street Journal, December 3, 2007. www.livemint.com, http://www.livemint.com/2007/12/03233411/Software-firms-reap-benefits-o.html

40 M. Meyer., Object-Oriented Software Construction, Second Edition, Upper Saddle River, NJ.: Prentice Hall PTR, 1997, p. 929.

sufficient evidence. This philosophy was also evident in the FAA AC20-148 Reusable Software Components guidance and by this framework.

Reuse involves taking advantage of previous successes and in order to continue, policies must support it and not overly reward the use of newly created software. If people are paid more to create their own software rather than reuse existing software and past successes, the outcome will be predictable. This suggests that an organization should have policies and processes that enable a learning organization and have supportive metrics and compensation policies for reuse. The organization with a reuse strategy and goal must utilize metrics that assess the progress towards that goal. These metrics, based on data and information gained from the reuse of software and on the impact of software reuse, should inform policy decisions, therefore providing feedback and an improvement mechanism for the organization. Aligning compensation to reuse or creating a learning organization, one that learns from its successes, means that delivering repeatable results are viewed more highly than one off results. Repeatable results built on the foundations laid by others should be viewed as more valuable than any single results alone.

These reuse processes and practices must be integrated into the existing software development process. For software reuse to permeate the system of systems, partnerships between stakeholders that will be promoted in policy and fostered by people are essential. These partnerships will be both intra- and inter-organization for software reuse to be a success. An example policy that will create and foster partnerships is a communication plan between the member organizations of the system of systems which will define the different types of communication, what they will be used to convey, and who is responsible for each.

Other interesting business practices where reuse is encouraged include software development companies offering discounts for the retention of intellectual property (IP) rights for the software developed for clients so that they can reuse it on their future projects without requiring completely new developments or violating IP rights. This encourages the company to reuse and to capture some of the original development cost of the software on future projects. It encourages software developers to design for reuse and to consider other potential users of the same software.

On the other hand, reuse is enabled through transparency and the delivery of required IP rights to the customer. An example of this is the U.S. Navy's open architecture (OA) strategy,[41] a multi-faceted strategy providing a framework for developing joint interoperable systems that adapt and exploit open-system design principles and architectures. This framework includes a set of principles, processes, and best practices that, among other things, achieves component software reuse. By requiring open architectures and hence transparency of components, they are more readily reused by the Department of Defense (DoD) and those who the DoD wishes to contract with. However, in order to achieve this, the USN will likely have to pay a premium for that transparency and quality of the software upfront and hence commit to reuse to avoid excessive costs overall on its programs. The Navy requires that on each project it obtains and retains flexible IP rights and the IP rights it agrees to be based on the rights being offered, the rights the Government wants or needs, and the potential reuse for that software. The IP rights obtained and retained should be dependent on where the reuse is to occur and who is likely to reuse the software. Open IP rights facilitates reuse as interfaces can easily be determined; however, IP rights can be limited but promote reuse as long as interfaces are known and the organization responsible for the component development is involved in the reuse effort.

The level of IP rights obtained and policy adopted in the acquisition of software will depend on the type of acquisition. In between these two approaches may lie a way to achieve a sufficiently open architecture while minimizing costs and allowing the software developer to keep the IP for subsequent reuse both within DoD and externally. It may be the case that subsequent reuse by a DoD software supplier is subject to government approval. Another approach maybe that the government could initially own the IP but if a reuse opportunity is available, the company applies to the government for approval with some potential royalties and discounts on maintenance (evolution) as a result being passed back to the government. What is required is a way to effectively control the software, encourage reuse, and minimize overall cost so that access is available to the right information when it is required.

41 Naval Open Architecture, October 2007, https://acc.dau.mil/oa.

The FAA's Reusable Software Components advisory circular AC20-148 provides a process for the regulator to review all relevant information for safety, regardless of whether the customer has all the information.[42] Maybe the DoD airworthiness regulators could implement a similar approach where they have access to all IP but the respective user does not necessarily, thus removing the requirement for the user to pay for it in order to ensure that it can be reused safely. This may, however, create a conflict of interest as the airworthiness regulator is often not independent from the end user of the software component.

Software for safety-critical system of systems should be developed with an assumption that it will be reused; however, it will not be blindly reused without considering its new context and performing a sufficient safety analysis. Policies and processes must be established that encourage and support reuse so that it is performed safely. Certain policies and approaches may be taken depending on who is reusing the software and for what purpose.

A DoD policy for software reuse in safety-critical system of systems is required similar to the FAA AC20-148 that encourages reuse in the right circumstances and provides guidance on how it is to be performed, leveraging previous successful software in new systems development. The elements of that policy should come from this framework providing guidance on how software reuse will occur and how credit can be gained to make future software and system development more effective and efficient.

8. Reuse Metrics

Frakes and Terry established many useful reuse metrics in 1996,[43] and only minor refinements have been made since then. The categorization of reuse metrics and models included cost-benefit analysis, maturity assessment, amount of reuse, failure mode

42 Advisory Circular AC 20-148, Reusable Software Components, US Department of Transportation, Federal Aviation Administration, December 7, 2004.

43 W. Frakes., C. Terry., Software Reuse: Metrics and Models. ACM Computing Surveys, 28(2), June 1996, p. 415-435.

analysis, reuse assessment and reuse library metrics. Lim[44] developed a framework for reuse metrics based on the same Goal-Question-Metric paradigm described earlier to divide reuse metrics into the following six types: Economic Metrics, Primary Metrics, Library Metrics, Process Metrics, Product Metrics, and Asset Metrics. The exact choice of metrics from those listed above will be based on the organization's particular situation and the questions it requires answers to, and their priorities. Ideally for reuse to be successful, it must be economical and lend itself to cost-benefit analysis and all other metrics will be subordinate.

9. Conclusion

Organizational factors affect the success of any reuse effort. Without organizational factors that support reuse and their integration into the software development process, the result of any reuse initiative is likely to fail. For reuse to be successful, and in particular in safety-critical system of systems, these organizational factors need to be aligned such that they support systematic and careful reuse. Focusing on the technical aspects of reuse and failing to address the non-technical or organizational aspects will condemn any large scale systematic reuse effort to failure. Policies should cover the conduct of software reuse and maximize the utility of software reuse in a systematic way.

C. REUSABLE SOFTWARE COMPONENT ATTRIBUTES

1. Overview

In consideration of those attributes the reusable software should possess, the software component is used, as it represents a common focus of reuse and a sufficiently sized entity to leverage the benefits of reuse.

44 W.C. Lim., Managing Software Reuse, A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components, Upper Saddle River, NJ.: Prentice Hall PTR, 1998, p. 301.

2. Definition of Software Component

A software component is a collection of software comprising a module with a well defined purpose that may be used with no or minimal alteration. A component is thus a bounded element in a design and can be a single unit within that design. A component includes both tangible (code, design, test plans, and documentation) and intangible (e.g. knowledge and methodologies) elements. Ideally, a software component represents a reusable piece of software that can be easily integrated with other components with relatively little effort. The key is to achieve that ideal level.

3. Component Attributes

The software component attributes described in this section represent the ideals or goals as it may not be possible to achieve all in the one component. The following component attributes may not be achievable to the desired level and the exact levels will depend on the results of trade studies. It is important that these attributes be identified early so that they can be included in the trade studies and given due consideration. Consideration of those desired attributes and incorporation of them into the development process represents a design-for-reuse approach. As mentioned earlier there are no standards for development of safety-critical software that maximizes the utility of software reuse.[45] This is also true in the safety-critical system of systems context. There is, however, some guidance on the certification requirements when an already certified software component is to be reused,[46] but this does not focus on the design for reuse or those inherent attributes that the software component must exhibit in order to support effective reuse. Thus, the focus here is on the attributes of reusable software components for use in safety-critical system of systems. That is, what attributes should be present to make component reuse easier or increase reusability of software in safety-critical system

45 J. Wlad., Software Reuse in Safety critical Airborne Systems, in Proc of 25th Digital Avionics Systems Conference, Oct 15, 2006, p. 6C5-1.

46 Advisory Circular AC 20-148, Reusable Software Components, US Department of Transportation, Federal Aviation Administration, December 7, 2004.

of systems. It is important that components be designed for reuse, in that they exhibit these attributes as they will become members of the repository of reusable software components.

Reusable software components for use in safety-critical system of systems should have the following attributes:

- They should be transparent to the user or reuser, exhibiting an open architecture by default, including the essential documentation requirements of that component (see §3D for details on how that transparency should be defined).

- The component should do one thing and do it well and have minimal interactions with other components. That is, the component should be highly cohesive with low coupling. Modular design should be adopted where functionality is partitioned into discrete, cohesive, and self-contained elements with well defined interfaces. This may be difficult to achieve in large component compositional reuse; however, this is a goal and modularity in design should be adhered to within the design of components.

- The software component shall consist of well formed modules, be protected, include the localization of data, and comply with the criteria and principles for modularity. Meyer's criteria for modularity—decomposability, composability, understandability, continuity and protection—and his five principles—direct mapping (Linguistic Modular Units), few interfaces, small interfaces (weak coupling and limited information exchange), explicit interfaces, and information hiding (providing uniform access)—should be followed in the design of reusable software components.[47]

Decomposability is satisfied if it is possible to decompose the software into a smaller number of less complex problems. This is synonymous with the "divide-and-conquer" approach to problem solving and the fact that it is often easier to construct a complex solution from the composition of a number of less complex solutions. The less complex problems should be connected via a simple structure with independence to facilitate further work proceeding separately on each one. If a component is

47 M. Meyer., Object-Oriented Software Construction, Second Edition, Upper Saddle River, NJ.: Prentice Hall PTR, 1997, § 3.1,3.2.

decomposable it will be easier to evolve because changes can be focused on the respective module. *Composability* means that it is possible to remove the software element from the environment of its original design and use it in another environment. *Understandability* means that the software must contain sufficient information so that it can be understood. This is also relevant for the evolution process, as in order to adapt and modify the software component it is essential that it be understandable. This is the key to component specification, which will be discussed in the next section. *Continuity* means that a requirements or specification change will only result in change of just one module or a small number of modules. *Protection* means that potential abnormal condition will be contained within a module or, at worst, will only propagate to a few neighboring modules.

Direct Mapping (Linguistic Modular Units) means that there should be structural correspondence (direct mapping) between the solution domain and the problem domain, as described by the model. In other words, a module should ideally perform a function or implement a feature from the problem domain description. The reusable software component should have as *few interfaces* and inter-communication as possible. The more relations and interfaces between modules, the more likely it is for the effect of a change or error to propagate to other modules. Those interfaces that do exist should be *small* and communicate as little information as possible (this is analogous to communication with limited bandwidth). Those interfaces that exist must be explicit and obvious from the information contained in each module. This interface information should be made explicit in the component specification. *Information hiding* or *uniform access principle* means that the implementation details of the services offered by a module should not be revealed but access to those services made available through a uniform notation.

- A reusable software component should follow the “open-closed” principle in that it will be usable as it stands (closed) while still being adaptable (open).

- The software components should be generic to allow for type variation and thus creating a wider range of reuse. It should be independent of implementation.

- The software component should include routine grouping in order to be self sufficient and cover all possible actions required for a particular purpose (i.e., include the complete set of routines required to achieve its intended purpose).

- The software component should cover a wide variety of implementations through a module family where it is not possible for a single module to satisfy all requirements, thus requiring a module / component family.

- The software component should be independent of representation; that is, the module should be able to carry out an operation without knowing implementation details and variants.

- The software component should factor out common behaviors. Instead of having a component for every type of implementation such as in a family of components, common behaviors can be factored out and modularized into a component reducing the number of components or modules in a family. This makes the reuse decision easier than having a different component for each type of implementation. This is a principle of modular design and is extended further by Aspect-oriented Programming (AOP) with its focus on separation of concerns.

- The component shall be robust in its environment. That is, it should react appropriately to a wide range of abnormal conditions when operating in the context of a system of systems. These reactions should be transparent as described above. Moreover, it should be complete, as opposed to efficient, and able to handle the conditions expected in the operating environment (safety-critical system of systems) with consideration given to unexpected but possible events. In an environment that is not safety-critical, it may be enough for the component to perform the desired function, and not be highly dependable. However, in safety-critical systems it is essential that the component is dependable, or in other words, from a safety perspective, failures will not produce hazardous conditions. In safety-critical system of systems the component must be complete and be adequately dependable (i.e., comply with the dependability requirements of the software system, derived from the system dependability requirements). Dependability is defined as:

> The trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers.[48]

Dependability consists of the attributes of reliability, availability, safety, and security. These complete components should include error handling, be fault tolerant, and satisfy the safety attributes of the dependability requirements of the software system. Measuring the degree to which a software component is complete for its intended operating environment is a difficult endeavor, especially given the size of the input and hazard space. To be complete with respect to requirements is different then being complete with respect to the environment. Only when the set of software component requirements is the same as the requirements for the context of intended use can completeness be truly measured, and it is very difficult to determine whether these sets are the same. The application of an effective requirements elicitation technique that captures context of use requirements including safety requirements can assist in ensuring that these sets are as identical as possible. Metrics can provide an indication of completeness, assuming that the requirements set, including safety requirements, is complete for the intended environment. These metrics will be described and discussed in §3F.

- The software component should be free of default behavior that the software engineer has not specifically described for that component.

- The component should be sufficiently evaluated and tested to verify the implementation of the safety attributes and requirements. Producer testing has been identified as one of the critical success factors for reuse.[49] Testing is essential in producing high quality software and for actually creating confidence in the claimed quality (the safety case will contain a significant contribution from testing). Even when one designs for quality and a certain safety attribute, it is still difficult to confirm or establish a case for that quality and attribute without testing. Testing must be performed to a level that builds the required safety case and meets the safety risk. Software systems

48 Dependability.org, IFIP WG-10.4, November 2006, http://www.dependability.org.

49 W.C. Lim., Managing Software Reuse, A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components, Upper Saddle River, NJ.: Prentice Hall PTR, 1998, p. 79.

safety testing must show the correct implementation of safety design requirements, verify safe implementation of safety-critical functions, and provide a basis for qualitative risk reduction in hazard analyses.[50]

- The software components must be designed with safety in mind, consisting of safety features and protection mechanisms, and should use formal semantics for specifying safety attributes and safety contracts. This could be performed by using a formal specification language to specify safety requirements and then designing to them. The use of formal methods is mandatory in the development of certain safety-critical software and their use here would be valuable in specifying the requirements in a precise, unambiguous form. For example, "The methods used in the SRS (safety-related software) development process shall include …: a) formal methods of software specification and design; …"[51]

- The component should comply with the important critical success factors for reuse quality of the component. If a reused component is not of sufficient quality, the success of any future reuse will be jeopardized. This comes back to the requirement that a reusable component should be of high quality and highly dependable and that quality or dependability be explicitly defined in terms of specific quality attributes (sometimes referred to as "-ilities") that the specific safety-critical context will determine. It is important to determine those attributes and define them in a meaningful and deliverable way. Applicable standards may be used to support the achievement of the requisite component quality; however, it is important to realize that using standards will not guarantee the required quality. Moreover, not using appropriate standards or processes will almost ensure that the desired quality is not present.

Safety contracts may be used to design components to be reused in safety-critical software. However, the application of these safety contracts is yet to be demonstrated. Furthermore, adequate tool support does not exist to ensure that these safety contracts are binding throughout the development process.

---

50 SW4582 Weapon System Software Safety Lecture Notes, Naval Postgraduate School, 2006.

51 UK Ministry of Defence, Defence Standard 00-55, Requirements for Safety Related Software in Defence Equipment, 1 Aug 97.

D. REUSABLE SOFTWARE COMPONENT SPECIFICATION

1. Overview

It is not enough to design a component for reuse and believe that it will subsequently be reused just because it exists and possesses the required attributes. Reuse must be enabled through appropriate specification of the component, providing the ability for a potential reuser to perform a specification match against the new requirements.

The goal of reusable software component specification should be to reveal the essential information about the component to allow an effective reuse decision to be made while concealing the nonessential. This is just like any other area in software engineering where the key is to find the right level of abstraction for the message you are trying to communicate. In defining what is required in the specification it is important to think of the user of the reusable software component and determine what information is required to make that reuse decision. Furthermore, the potential user of the software component must be able to locate a candidate component. The question to be answered is: "Does the software component meet my requirements and what are the risks?" In safety-critical system of systems the answer will not be simple and will involve the composition of many different types of information on the component. What will be required is the ability first to identify potential reuse candidates based on how close the specification meets the requirements and then to confirm the requirements match through further analysis.

2. Software Component Specification

There are many ways to specify a software component and completely describe it to provide the desired level of understandability. The level of understandability for safety-critical system of systems components is relatively high. Early work on specifying software components included the 3C model by Weide et al.[52] and Tracz,[53] which was a highly abstract framework model:

52 B.W. Weide., et al. (1991) Reusable software components, Advances in Computers, Yovits, M. C (ed.), Vol. 33, Academic Press.

53 W. Tracz., The 3 Cons of Software Reuse, in Proc of the Third Annual Workshop on Software Reuse, July, Syracuse, NY, 1990.

Concept. A statement of what a piece of software does, factoring out how it does it (abstract specification of abstract behavior).

Content. A statement of how a piece of software achieves the behavior defined in its concept (the code to implement a functional specification).

Context. Aspects of the software environment relevant to the definition of concept or content that are explicitly part of the concept or content.

Under that model, it could be still quite difficult for a user to find the component sought. Another model has been suggested to describe the software component and increase the transparency of the information of a particular software component through an extension of the Class Responsibility Collaboration (CRC) cards used in object-oriented design. This model, proposed by Riehle,[54] also at a high level of abstraction, can be used to describe the component providing more relevant information for matching than the earlier models of Weide and Tracz.

The CRCCC or CRC^3 specification as described by Riehle[55] at the component level is:

- **Component Identifier**. Self-describing name.
- **Responsibility.** What problem are we trying to solve?
- **Collaborations.** Identify other components.
- **Constraints and Bounds.** Design metrics for each element.
- **Controls.** Prevent internal and external variations, behaving like a feedback control loop with a (possibly varying) set-point. This also involves a clear statement of the action to be taken when a control is violated.

54 R. Riehle., Software Design Metrics: Designing Software Components to Tolerances, ACM SIGSOFT Software Engineering Notes, Volume 32, Issue 4, Article 7, July, 2007, p. 1-6.

55 Ibid., p. 4.

Extending on the specification models described above we propose a new specification model to provide the necessary information for specification matching in safety-critical system of systems. That model, the C^5RA model, is described as follows:

- **Component Identifier.** This is the name of the component, which should be semantically accurate and conform to an agreed ontology convention.

- **Collaborations.** This specifies any other components it requires to collaborate with to solve the problem utilizing the same ontology for the component identifier (as per section 3C, the design goal is to keep collaborations to a minimum). Furthermore, it is to include interface specifications for those collaborations.

- **Constraints and Limitations.** The constraints and limitations are expressed in terms of design metrics for each element, including those related to safety, quality attributes, pre and post conditions, and invariants.

- **Controls.** The controls for the software component should describe the failure conditions, safety features, protection mechanisms, and any potential safety concerns, as well as the mitigation strategy. The controls are the means for preventing failure and for mitigating the hazard causal factors.

- **Configurations.** This is a description of the set of possible internal software configurations for the component.

- **Responsibility.** This is a description of what problem the software component will solve and will typically be expressed in functional terms.

- **Analysis.** This is the analysis of all safety concerns for the component, including interface safety concerns. It also consists of those open problem or safety reports, any assumptions made in the design of the component, and any supporting data or relevant information that supports the safety case or possible certification.

This model has a different definition of control to the Riehle model.

This specification model is required in addition to the set of software artifacts that comprise the software component in order to provide a complete specification of the known information on the software component. The set of software artifacts would be

provided separately to support further analysis and integration of the component into the system and system of systems post selection. This organizes the information on a software component in a suitable form to enable specification matching. The C^5RA specification model partitions the relevant information into separate descriptive categories that can enable the initial search and, along with the information contained in each of the software component artifacts, represents a complete collection of the information for a component. It is important that each descriptive category provides sufficient detail to achieve the initial match and that software component information is available to conduct further analysis in order to downselect on the most appropriate component. This makes it easier to identify for reuse with a more complete understanding of what it will do.

To complete the C^5RA specification of a software component, an accompanying ontology is required so that proper classification occurs including naming conventions and relationships between entities (composition and decomposition relationships), which will reduce duplication and aid in searching (specification matching). The ontology will be organizational and domain specific and should be contained in policy and managed by the reuse librarian as well as the managers of those areas within the organization developing reusable software components. A standard ontology among an industry would enhance the ease of specification matching to requirements and facilitate more effective reuse. This would require commitment from industry or system of systems regulators and mandate that component specifications follow the C^5RA format and a descriptive ontology. For example, the FAA in AC20-148 requires a number of deliverables, as described in DO-178B, to be submitted and could request that information is provided in accordance with the C^5RA format using a standardized ontology to provide a high level abstraction to aid their analysis. This could also be utilized by other regulators such as the Naval Air Systems Command for USN and USMC aviation and the ADF for military aviation in Australia. For example, within the aviation industry, operating safety-critical system of systems the high level choices in the ontology for potential component identifiers within avionics software components may be: aircraft management, mission computing, navigation, collision avoidance, radar, sensor systems, data links, diagnostics,

weapon management, and weapons. The collaborations section of the C5RA specification may also use the same ontology when referring to the components it collaborates with to solve the problem as well as the type of collaborations.

The component shall be specified in a language and description model (such as UML) that is in demand, standardized, and widely understandable. This is significant in this application domain as certain languages are better suited to conveying information about safety than others but it will be dependent on the specific domain of the system of systems and what the industry standard specification language may be, including formal specification languages.

Complete documentation of the software including the C5RA is important to ensure transparency, understanding, and to support an effective reuse decision in safety-critical system of systems. This is required of all reusable software components as the code itself only contains approximately 10% of the information on the software.[56] The C5RA abstract model will not contain all relevant information for all potential contexts of use but it represents a sound approach to capturing that information that is known in the current context and providing it for potential users of the reusable component. Potential future users may require more information, depending on the exact context of use. Major design decisions should be recorded in their respective artifacts with a complete description of the rationale. The metrics should be for software quality and efficiency and be specific to the component's functionality or potential use. It is therefore essential that for reusable software components to be used in safety-critical systems, software quality metrics be documented (third C in C5RA specification). Metrics providing indicators of quality attributes would be applicable to the reusable components. Software quality metrics are essential for reusable software components with potential use in safety-critical systems.

All information about the component should be part of the component itself. Some languages support automatic documentation, increasing the reusability of the software (e.g., RDoc in Ruby). The documentation on the reusable software component

56 M. Auguston., SW4540 Software Testing Lecture, Naval Postgraduate School, Monterey, March 26, 2007.

should also contain information on certifications achieved (e.g., FAA certifications for airborne software components, especially what level of certification has been achieved for the particular registered reusable software component) and any open problem reports. It is particularly relevant to record whether in the case of civil aviation, the FAA has accepted a software component as an RSC (Reusable Software Component). This acceptance information will provide potential users of the software component with information of its certification baseline and what may be required for their particular context of use.

The C^5RA model represents an effective component specification scheme at a high level of abstraction; however, all information of the software component should be available and contained with the product itself. C^5RA represents a good starting point that, when accompanied by an ontology and division of component information within, can aid in locating candidate components for reuse and thus finding a specification match for the requirements.

E. SAFETY PROCESS AND HAZARD ANALYSIS

1. Overview

A component that has been designed for reuse, meeting the attributes of reusable software components, and is specified appropriately (making it easy to identify and locate; and hence perform specification matching) does not automatically make it suitable for use in safety-critical system of systems. Safety-critical system of systems are so complex[57] that each software component must be assessed for suitability in the context of use through a hazard analysis or, in this specific case, a system-of-systems hazard analysis. This is further reinforced in the FAA's air circular guidance (AC20-148) on reusable software components (RSC) where for acceptance as an RSC and each subsequent reuse requires that there be no safety and other concerns before approval, although it is not specified how this will be done.

57 The reasons for this complexity are largely as a result of their not being one controlling entity over all within, the fact that the systems were not usually designed to operate in a system of systems and that there are many possible configurations and interfaces between concomitant systems.

Having quality well specified software components does not guarantee safe operation in an environment and state space that is significantly large and contains additional hazards to those represented by the sum of all concomitant system hazards in the system of systems. These additional hazards are called emergent hazards and need to be identified, analyzed, and treated along with revisiting and analyzing those existing system hazards. An emergent hazard is a hazard that may occur within a system of systems that is not attributable to a single system.

A system-of-systems hazard is any hazard that may occur within a system of systems. The system-of-systems hazards or hazard space consists of those hazards attributable to the individual systems of the system of systems and emergent hazards. A system-of-systems hazard analysis is essential before an effective reuse decision is made on a software component for use in a safety-critical system-of-systems. This is required once a specification match is achieved and the software component incorporated into a system. When a software component is considered for reuse in this environment, the system-of-systems hazard analysis must be updated including the analysis of the system hazards as well as any potential emergent hazards that could arise through the use of the system containing the reused software component. The reassessment of system hazards as a result of the new component is defined and documented in MIL-STD-882D and the system safety handbook;[58] however, the emergent hazard analysis process is relatively immature. The identification of system-of-systems hazards involves a collaborative effort on behalf of all program members and stakeholders in the system of systems, just as it does in the identification of system hazards; however, in this new context that collaboration requires effective management to ensure a comprehensive set of hazards is identified and effectively treated. A system-of-systems hazard ontology is required to assist in considering all potential hazard types, providing consistency and for providing a starting point for the emergent hazard analysis part of the system-of-systems hazard analysis. A detailed ontology will provide an abstract description of the hazard type and

58 National Aeronautics and Space Administration. System Safety Handbook (DHB-S-001). Dryden Research Flight Center, Edwards, CA, 1999.

assist in ensuring that all potential hazard types are investigated and analyzed to the extent possible given project constraints and the system of systems safety requirements.

When a decision is made to reuse software in safety-critical system of systems the system-of-systems hazard analysis needs to be updated. It is possible to identify system hazards as a result of the reuse in the system(s) containing the reused software following the process defined in MIL-STD-882D or adapting the AS/NZS 4360 risk management standard; however, not all hazards will be considered using that process alone. The system hazards in the new context following reuse will likely involve those from the previous system hazard analysis and will provide a good starting point for the new hazard analysis. The goal in the case of reusing software is to update the system-of-system hazard analysis by updating the system hazard analysis and the emergent hazard analysis. That update may include new system hazards and emergent hazards that must be identified in the first instance.

The system safety process and the risk management process are very similar, both starting with the identification of their respective elements followed by analysis and a way of treating those elements. The risk management process identified in AS/NZS 4360 is more iterative, involving the monitoring, reviewing, communicating and consulting of risk information throughout the process than that defined in MIL-STD-882D. It is the identification part in the process that is the most difficult because it is often not easy to identify all risks, hazards, or potential risks and hazards. If the risks or hazards are not identified, then there is nothing that can intentionally or consciously be done to treat them. Furthermore, it is often the identification part of the process that is the entry point for risks or hazards and if missed here, they will not be considered. This is further exacerbated in a system-of-systems context because there are emergent hazards that cannot be attributed to any one system in isolation. Therefore, a technique that just sums those system hazards within the safety-critical system of systems will omit an important and significant category of hazards. Ways to overcome the identification problem are to:

- Use a well developed ontology of hazards or hazard categories and apply a hazard analysis process to each category. This will serve to identify the high level abstract hazards that could possibly exist and allow for exploration in more detail in each category, through hazard decomposition, considering the impact of the addition of the new component.
- Brainstorm or use the collaboration of all stakeholders as a potential source of hazard information and empower them to identify hazards in their own area for consideration by the system safety team.
- Utilize safety experts. This is required in most system safety efforts. System safety experts or team members are essential members of the development and operational team through the input of safety information into the process; however, they do not provide a systematic way of identifying all hazards. Furthermore, they do not necessarily have expertise in areas where hazards may exist. They are not the panacea to the identification problem but serve as an essential member and part of the solution.

It is the hazard identification problem that is the most important because once identified, hazards can be considered and tracked and the appropriate treatment provided in the right form at the right time.

When a software component or artifact is to be reused, the system-of-systems hazard analysis should be revisited and updated to include the new component and its system. One should also revisit the emergent hazards analysis and consider the emergent hazard categories for new hazards.

A candidate process for conducting system-of-systems hazard analyses that incorporates software reuse is:

1. Establish the context or environment for the system of systems.
2. Identify all systems in the system of systems.

3. Conduct the system safety process as defined in MIL-STD-882D or similar standard such as AS/NZS 4360 to identify, analyze, evaluate, and treat all system hazards including those of the system(s) containing the reused software.
4. Define the system-of-systems architecture.
5. Identify the emergent hazards of the system of systems using a suitable technique based on a ontology of hazards (such as those partially identified by Redmond[59] to include reconfiguration, integration, and interoperability hazards at a high level of abstraction) to assist in considering all the potential hazard types and their specifics based on the system-of-systems architecture. It is essential to utilize stakeholder collaboration and safety experts as part of this process.
6. Analyze emergent hazards.
7. Evaluate the mishap risk.
8. Treat the mishap risks.
9. Evaluate the residual mishap risk (and compare with the acceptable risk).

The process involves continual feedback, review, monitoring, communication and consultation of hazard and risk information and continues until the mishap risk reaches an acceptable value determined by the appropriate governing body (or regulator) of the system of systems. Furthermore, the process is iterative in nature and not entirely sequential. Step 3, the system safety process, should be performed in parallel with steps 4, 5, and 6 with collaboration and communication between them to ensure sharing of relevant hazard information. This process involves the monitoring, reviewing, communicating, and consulting of risk information throughout the process as per the AS/NZS standard.

59 P. Redmond., A System of Systems Interface Hazard Analysis Technique, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 2007, p. 33.

F. METRICS

1. Overview

An important part of the information about any software component and program activity are the relevant metrics. This section introduces those metrics of interest when developing software for reuse in safety-critical system of systems and those metrics relevant to measuring the success of a reuse program. In the C^5RA specification model, these metrics should be made available to enable future reuse (as well as evolution) and should be specified within the Constraints and Limitations element of the model.

2. Software Safety Metrics

To minimize risk (safety or otherwise) and increase the chance of a successful outcome, quantitative support for management decision making is required; that is provided through effective metrics. Metrics are an extension of measurements when provided within context, enabling greater interpretation and understanding of what is occurring. They are indicators that provide the impetus for control of what is of value.

Software safety cannot be proven or predicted due to environmental uncertainty and because exhaustive testing is infeasible. Even if all potential hazards are identified, one can only demonstrate that you have not found the software exhibiting any behavior that could result in a hazard, not that the software is free from conditions that will result in a hazard. Formal methods, which are the software development activities that employ mathematically based techniques for describing, reasoning about, and realizing system properties, expressed using formal languages[60] will not reveal the absence of a safety requirement. Software safety metrics will not tell us whether the system is safe, but they can provide indicators of potential safety problems and risks. Software safety risk metrics are intended to support detection and analysis of software-related safety issues in a timely manner and make potential risks visible. Furthermore, they can be used in developing the

60 D. Drsinsky., J.B. Michael., M. Shing., The Three Dimensions of Formal Validation and Verification of Reactive System Behaviors, Naval Postgraduate School Technical Report NPS-CS-07-008, Monterey, CA, August 2007.

safety case for the software and in providing quantitative evidence to regulatory authorities. The organizational use of metrics can be characterized by the CMMI process maturity level of that organization.

Most of the work in deriving software safety metrics is based on the Goal, Question, Metric (GQM) methodology developed by Basili and Weiss, which uses a general process to determine effective metrics. That general process involves:

1. Identification of information needs.

2. Interpretation of an information need as being within an information category.

3. Identification of measurable concepts within each information category.

4. Identification of prospective measures, associated with each measurable concept. [61]

This approach to determine effective metrics ensures the rationale behind the metric exists and that what is subsequently measured has value and forms an aid to effective decision making and control.

Reliability, an important software quality metric and sub-element of dependability, can be used as an indirect indicator of safety, with caution. Although orthogonal dimensions of dependability, they are often erroneously equated. Reliability is defined as the probability that an item will perform a required function, under stated conditions, for a stated period of time.[62] Safety is the probability that conditions that can lead to a mishap (hazards) do not occur, whether or not the intended function is performed. In general, reliability requirements are concerned with making a system failure free, whereas safety requirements are concerned with making it mishap free. These are not congruent goals and are therefore not synonymous. Reliability is concerned

61 V. Basili., D. Weiss., A Methodology For Collecting Valid Software Engineering Data, IEEE Transactions on Software Engineering, October 1984.

62 D.J. Smith., Reliability Maintainability and Risk, practical Methods for Engineers, Seventh Edition. Oxford, England.: Elsevier Butterworth-Heinemann, 2005, p. 12.

with every possible software fault,[63] whereas safety is only concerned with those faults that may result in actual system hazards. Not all software faults cause safety problems, and not all software that functions according to specification is safe. Severe mishaps have occurred while something was operating exactly as intended, that is without failure. Improving reliability will not necessarily improve safety. It will depend on where the reliability improvement is made. Improving reliability by removal of faults or preventing the propagation of faults may not remove or prevent propagation of those faults that lead to safety related failures and thus not improve safety at the same time. If, however, all safety requirements are in the software specification, then reliability can provide a more accurate predictor or indicator of safety. If reliability is improved in the area of safety-critical software and all safety requirements are included, an increase in safety will result. Reliability of the software component and its system is important; however, it should be monitored and analyzed separately from safety, when ensuring system dependability requirements are achieved.

The identification and implementation of safety requirements is a key function in system and software safety. All software safety requirements are to be met in the design and implementation of the software and must be verified: each safety requirement must have a test case or cases associated with them. For verification there needs to be a mapping between the safety requirements and test cases with this mapping reflected in the Software Safety Requirements Traceability Matrix.[64] A potentially useful metric to determine whether verification activities have been identified for the safety requirements is the number of safety requirements unlinked to test cases (SRUTC).[65] If this value does not tend to approach zero over time, then the safety risks are not being identified as requiring testing or verification.

---

63 Fault in this case refers to a defect within the system.

64 J.B. Michael., SW4582 Weapon System Software Safety Lecture Notes, Naval Postgraduate School, 2006.

65 V. Basili., K. Dangle., L. Esker., F. Marotta., Gaining Early Insight into Software Safety: Measures of Potential Problems and Risks, in Proc of the Systems & Software Technology Conference, June 2007, p. 22.

If f(x) is the Safety Requirements Unlinked to Test Cases (SRUTC) and x is time to release of the software, then:

$$\lim_{x \to 0} f(x) = 0$$

Equation 1: Desired Value of SRUTC Over Time

SRUTC is most useful during the requirements analysis and design phases where it is important that a means of verification has been established for each safety requirement. SRUTC can ensure that when a safety requirement is identified and analyzed, it is written in a manner that aids testing and plans for proof of it being met.

Another similar metric that we propose that is available later in the system lifecycle is the safety requirements demonstration metric (SRDM),[66] which is obtained by dividing the total number of separately identified safety requirements in the software requirements specification (SRS) that have been successfully demonstrated by the total number of separately identified safety requirements in the SRS. Ideally the SRDM should be equal to one, indicating that the implementation of each safety requirement has been tested.

$$\text{SRDM} = \frac{\text{Total Number of Demonstrated Safety Requirements}}{\text{Total Number of Safety Requirements}}$$

Equation 2. Software Requirements Demonstration Metric

SRDM differs from SRUTC in that the former monitors the demonstration of the safety requirements whereas the latter ensures that the verification activity has been identified.

To place the required emphasis and priority on safety, it is essential in the development of safety-critical software that an adequate proportion of the requirements set consists of safety requirements. Thus the question that could be asked here is: "Are there a reasonable number of software safety requirements being identified?" The metric

66 Adapted from the requirements demonstration metric in IEEE standard for Software Quality Assurance Plans, 730-1998, p. 3.

could be Percent Software Safety Requirements (PSSR),[67] which is the percentage of software safety requirements relative to the total number of software requirements. If the number of identified software safety requirements is not "reasonable" relative to the platform family or in line with system safety in general, this would represent a risk. If there are too few software safety requirements, this would represent a safety risk and alternatively too many could result in cost and schedule risks for the project or development. The assessment of reasonableness would be based on heuristics derived from past experience in the development of safety-critical software and on engineering judgment of what would make a reasonable percentage given the potential hazards and risks. This metric has the most relevance during requirements and analysis activities of the software life cycle, where an early indication of whether safety is being adequately considered in the development is available.

$$\text{PSSR} = \frac{\text{Number of Software Safety Requirements}}{\text{Number of Software Requirements}}$$

Equation 3. Percent Software Safety Requirements

Other safety metrics of value are related to hazard identification and provides answers to the following questions:

- Have a reasonable number of software safety hazards been identified?
- Are causes, controls and verifications being generated over time?
- Does every cause have at least one control?
- Does every control have at least one verification to test that the control has been implemented?

The subsequent metrics[68] are:

67 V. Basili., K. Dangle., L. Esker., F. Marotta., Gaining Early Insight into Software Safety: Measures of Potential Problems and Risks, in Proc of the Systems & Software Technology Conference, June 2007, p. 17.

68 V. Basili., K. Dangle., L. Esker., F. Marotta., Gaining Early Insight into Software Safety: Measures of Potential Problems and Risks, in Proc of the Systems & Software Technology Conference, June 2007, p. 18.

• Percent Software Hazards (PSH), which is the number of software safety hazards divided by the number of system safety hazards. If number of identified hazards is not "reasonable" then there are problems with the hazard management and analysis process for software.

$$PSH = \frac{\text{Number of Software Safety Hazards}}{\text{Number of System Safety Hazards}}$$

Equation 4. Percent Software Hazards

This metric, like the Percent Software Safety Requirements, will indicate whether software hazards are receiving the appropriate attention compared to all system safety hazards. This metric also has most relevance during requirements and analysis activities of the software life cycle to provide early consideration and indication of whether software safety hazards are being appropriately identified in order for them to be managed. This metric should also be updated throughout the software life cycle to keep the safety case current and to ensure software safety hazards are being appropriately considered when the system and software undergoes changes and evolution.

• Controls with causes (CwC) is the number of software safety hazard causes for which there is a control divided by the number of causes for all software safety hazards. If there are causes without controls then there are problems with the hazard management process for software. This metric is most relevant during the analysis, design, and implementation phases, where software safety hazard controls are designed into the software component and system.

$$CwC = \frac{\text{Number of Causes with a Control}}{\text{Total Number of Causes for all Hazards}}$$

Equation 5. Controls with Causes

• Verifications with controls (VwC) is the number of controls for which there is a verification divided by the number of controls for all hazard causes. If there are controls without verification, then there are problems with the hazard management and analysis process for software.

$$\text{VwC} = \frac{\text{Number of Controls for which there is a Verification}}{\text{Total Number of Controls for all Causes of Hazards}}$$

Equation 6. Verifications with Controls

This metric follows on from CwC, where it is essential for software assurance that the hazard-cause control is associated with an appropriate verification. This metric is most relevant during the analysis, design, and implementation phases, where software safety hazard controls will be linked to verification activities. These verifications will subsequently guide testing of the software component to ensure that the controls implemented perform as intended and reduce the safety risk.

Another metric identified by Basili et al. within hazard management that can be a useful indicator of safety answers the question: "Is the number of open software hazard components (causes, controls, and verifications) shrinking over time?" What is desirable is for the number of open software causes and controls to be closing over time to indicate that hazards are progressively being brought under acceptable control. This metric, called the Hazard Cause/Control Closure Evolution (HCCE),[69] is a three point moving average of the set of open causes and controls at three consecutive time intervals. This metric indicates relative performance of closing hazard causes with controls and verification with respect to previous time periods (rather than just looking at the number of open causes in isolation). If greater than one, this would suggest that hazard causes or controls are opening faster than they are closing. What is important here is to understand the relationship between the metric and the stage in the lifecycle the software is in. Early on values greater than one are acceptable; however, at some point the value should remain below one. The value will also depend on the type of system developed and its environment, although at some stage it would be expected that hazard causes or controls will be opening faster than they are closing (early in software life cycle). A decline in this value over time will suggest that hazard causes and their controls are being brought under control and be representative of system maturity and context of use stability.

69 V. Basili., K. Dangle., L. Esker., F. Marotta., Gaining Early Insight into Software Safety: Measures of Potential Problems and Risks, in Proc of the Systems & Software Technology Conference, June 2007, p. 19.

Another metric identified by Basili et al. determines whether the software being developed is receiving the appropriate level of rigor for its software risk. Level of rigor is defined as:

> The amount of requirements analysis, development discipline, testing, and configuration control required to mitigate the potential safety risks of the software component.[70]

Example level of rigor metrics are Percent Requirements Level of Rigor (PRLOR) and Percent Code Level of Rigor (PCLOR),[71] which compares whether the level of rigor employed is what is expected of software with a certain level of autonomy or control categorization. This is similar to ensuring that the software receives the appropriate level of analysis and testing for its control category according to the software hazard risk index within the software risk assessment matrix. An example software level of rigor and what is expected at each software level is contained in Appendix A. Another example of a similar approach is the software life cycle process objectives and outputs by software level provided in DO-178B Software Considerations in Airborne Systems and Equipment Certification.[72] It is important to have a metric that indicates whether the appropriate rigor or attention is given to the category of software. That metric can be expressed in percentage terms against what is expected. If these percentages are not within reasonable values, then safety-critical software might not be developed to the appropriate level nor receive the appropriate level of attention during development. Additionally, if the percentage is too high, this could increase the cost and schedule risk for the project. This type of metric is relevant throughout software development and can be used during all activities in the development of software for safety-critical systems. This metric also appears to be applying a CMMI type approach to the development of safety critical software and ensuring that certain activities occur depending on the level of software being developed. There is an underlying assumption in this metric that the

70 V. Basili., K. Dangle., L. Esker., F. Marotta., Gaining Early Insight into Software Safety: Measures of Potential Problems and Risks, in Proc of the Systems & Software Technology Conference, June 2007, p. 33.

71 Ibid., p. 21.

72 Radio Technical Commission for Aeronautics, DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification, Washington D.C., 1 December 1992, Annex A.

development effort should depend on the level of criticality of the software. There should also be an industry wide accepted best practice for what that effort should be when developing safety-critical software. As described later in this section, the +SAFE[73] extension to the CMMI provides an example of this type of approach.

It is also essential to determine whether software safety defects are being resolved. A relatively simple metric can be applied here for safety defects that is a measure of the number of safety defects and then a list of them in priority order. This will assist in the progress of resolving safety defects and provide an indicator of safety for the software. Furthermore, safety defect density can indicate how effectively the development designed software that was free of safety defects. This metric could also be used to compare projects and create a baseline for organizational process improvement in the removal of safety defects. This metric is applicable throughout verification and testing where the defects are identified and the test results used during both redesign and re-coding efforts.

Another approach to minimizing risk including safety risk is the application of the SEI CMMI. The CMMI characterizes an organization based on the maturity of its processes on a scale of one to five, from initial (1) to optimized (5). The premise underlying CMMI is that improved and mature processes result in higher quality and risk reduction. Contracting with an organization at a higher maturity level on the CMMI model reduces development risk and increases the chances of a successful outcome. The Defence Materiel Organisation (DMO) in Australia, users of the CMMI to improve its acquisition and maintenance of software-intensive systems, recognized that the model may inadequately address the specialized needs relating to safety-critical systems and so developed a safety extension in conjunction with the Software Verification Research Centre (SVRC) to the CMMI called "+SAFE".[74] The aim of this extension is to identify

73 Software Engineering Institute, Defence Materiel Organisation Australian Department of Defence. +SAFE, V1.2 A Safety Extension to CMMI-DEV, V1.2, Technical Note CMU/SEI-2007-TN-006, March 2007.

74 Software Engineering Institute, Defence Materiel Organisation Australian Department of Defence. +SAFE, V1.2 A Safety Extension to CMMI-DEV, V1.2, Technical Note CMU/SEI-2007-TN-006, March 2007.

the safety strengths and weaknesses of product and service suppliers, and to address identified weaknesses early in the acquisition process.[75] The safety extension was developed so that CMMI appraisers and users can become familiar with the structure, style, and content provided to reduce dependence on safety domain expertise.

This extension to CMMI consists of two safety process areas added to CMMI-DEV to provide an explicit and focused basis for appraising or improving an organization's capabilities for providing safety-critical products. Those two safety process areas are Safety Management within the Project Management CMMI category and Safety Engineering within the Engineering CMMI category. It essentially provides a process for developing safety-critical products, and a metric or measurement would be how well does the organization's process for developing safety-critical software follow that defined by the +SAFE extension. In other words, the metric would be representative of and answer the question of "how close to this benchmark is the organization?" The metric used is the integration of the percent satisfaction of the specific goals within each process area and CMMI category. The closer the organization is to the +SAFE process the greater the organization's understanding of the safety domain and the increased likelihood that the organization will be able to deliver safe software. This information would not only be useful in the development of safety-critical software, but it also provides evidence for input to the safety case for the software (and the parent system) and assists in making assessments of an organization's capability for developing safety-critical software.

The work of Jones proposes a key node safety metric and a safety improvement algorithm. The key node safety metric predicts the relative safety between different versions of software modules using a heuristic analysis of fault tree structure and calculates a value based on the fault tree properties such as key node height, size of key node sub-trees, and the number of key nodes. The safety improvement algorithm provides an objective method of improving a system's safety by determining which components

75 M. Bofinger., N. Robinson., P. Lindsay., M. Spiers., M. Ashford., A. Pitman., A. Experience with Extending CMMISM for Safety Related Applications, in Proc of the 12th International Symposium of the International Council on Systems Engineering, (INCOSE'02), Las Vegas, Nevada, 2002, p1.

need improvement and by what amount in order to achieve the desired increase in safety. The algorithm also provides an estimate of resource requirements in man-hours to meet the safety requirement. The key node safety metric has much promise and benefit as it is based on causal factors and the relationships between events that result in hazards. Moreover, as during hazard analysis, fault trees are prepared incorporating this metric would be easy as it could leverage off the existing fault trees and involve defining the properties of those already identified key nodes. This metric has potential value in enabling decisions on safety of designs, minimizing the potential for causal factors to propagate, and in building in fault tolerance. This metric requires testing on whether it benefits the design and safety assessment of a safety-critical software-intensive system. With this metric, a pilot project should be used to validate it and to test whether it effectively supports decision making. If this metric is validated, it could have a significant impact on the evolution and improvement of software components in safety-critical system of systems.

Test metrics are also relevant to software safety assessments. The test coverage and results will be a source of metrics information to determine the safety of the software. The relevant metrics include test case criteria, such as statement, decision, condition, and define-use pair and usage-based coverage. It would be preferable to conduct higher criteria testing on the safety-critical software and that software that has successfully passed those higher criteria testing can be inferred as being safer. The level of testing and its coverage will depend on the resources available and the requirements of any regulatory agency. The degree of test coverage of safety critical software should be adequately determined. If software testers are aware of the need for additional test coverage of safety critical functions, these will be incorporated into the routine testing which will then influence developers to design in safety as they will be aware of the increase in quality required of their software.

Although the above software safety metrics appear to be effective in theory, they should be validated in a realistic environment. A number of the metrics also rely on subjective assessments, baselines, or heuristics to determine what is a reasonable value for the metric, so the application of these may take time and a number of developments to

achieve optimal outcomes (i.e., determination of what "reasonable" values are). These software safety metrics add value to the development and provide an important indicator of software safety and support management decisions. They are also important safety indicators for designing reusable software components for use in safety-critical system of systems.

It is important not to act on metrics information alone, to understand that they are indicators of potential safety issues, and to use them as triggers for further analysis, which should be inherently available in the component. In the application of metrics it is important that the essential ones be implemented (those that provide the best indication of safety) and the collection methods resourced. It is essential to focus on the key indicators and not try to collect as much information as you can as this will only distill that information that is critical. This needs to also be balanced with the measurements already conducted so that the safety metrics collection process integrates seamlessly with other metrics processes.

There are other metrics, such as those identified in IEEE Standard for Software Quality Assurance Plans, 730-1998 (branch metric, decision point metric, domain metric, error message metric) which provides minimum acceptable requirements for preparation and content of Software Quality Assurance Plans (SQAPs) applicable to the development and maintenance of critical software; however, they may relate more to the aggregation of quality attributes as opposed to the more safety focused metrics described here. Furthermore, this standard provides the minimum requirements for SQAPs and would thus be expected to be applied by most organizations developing and maintaining critical software. What is required are metrics that are more focused on safety and descriptive of the safety of the software.

3. Summary of Software Safety Metrics

Table 3 represents a summary of the software safety metrics identified herein, their relative ranking of importance (usefulness) in designing safe software and the part of the software lifecycle where they are most relevant.

Rank	Software Safety Metric	Description	Expected & Preferred Values	Software Life Cycle Activity with most Relevance
1	Percent Software Safety Requirements (PSSR)	The number of software safety requirements divided by the number of software requirements	Reasonable Value based on heuristics and experience	Requirements
2	Percent Software Hazards (PSH)	The number of software safety hazards divided by the number of system safety hazards	Reasonable Value based on heuristics and experience	Requirements
3	SEI CMMI +SAFE Extension	A process for developing safety critical products and a measure of how close to the standard a developer is	100% meaning that the organization complies with all aspects of the preferred process	Assessed before development begins
4	Percent Level of Rigor (or analysis and testing effort)	Compares the level of rigor employed to what is expected of software with a certain level of autonomy or control categorization	100%	Design and Implementation
5	Software Requirements Unlinked to Test Cases (SRUTC)	The number of safety requirements unlinked to test cases	0 or approaching zero	Requirements, Analysis, and Design
6	Open Software Safety Defects	The number of open software safety defects with priority	0	Design, Implementation, and Testing
7	Controls with Causes (CwC)	The number of causes with a control divided by the total number of causes for all hazards	1	Design and Implementation
8	Verifications with Controls (VwC)	The number of controls for which there is a verification divided by the number of controls for all hazard causes	1	Design, Implementation, and Testing
9	Hazard Cause/Control Closure Evolution (HCCE)	Three point moving average of the set of open causes and controls at three consecutive time intervals	1 or less. Greater than 1 means hazard causes or controls are opening faster than they are closing	All activities. Design and Implementation
10	Software Requirements Demonstration Metric (SRDM)	The total number of demonstrated safety requirements divided by the total number of safety requirements	Reasonable Value determined by past experience	Testing
11	Test Metrics	Test case criteria such as statement, decision, condition and define-use pair and usage-based coverage and the results	Type of test coverage and a successful test	Testing

12	Key Node Safety Metric (S)	A metric for predicting the relative safety between different versions of software modules using heuristic analysis of fault tree structure, based on fault tree properties such as key node height, size of key node sub-trees, and the number of key nodes	0 to 1. The higher the value the better	Design and Implementation

Table 3. Software Safety Metrics Ranking and Relevance

G. REGULATOR NEEDS

As explained in Chapter II, the regulator will have significant influence on the elements of the framework and what evidence is required for each. This will depend on the specific regulator, domain of operation, and the public interest as most regulators are independent public bodies established to protect the public interest and public safety. They will exert influence and help shape the specific requirements within each element of the framework as described in this chapter.

H. SUMMARY

The framework described herein prescribes that in order to successfully implement software reuse in safety-critical system of systems, organizational factors must be supportive, components must be designed for reuse, those components sufficiently specified, and an assessment on the environment suitability made before deployment. The metrics covered in this chapter support the safety aspects of the software and those that are related to measuring the success of the reuse itself. The regulator will be expected to further shape the requirements of each section of the framework and specify the form that evidence is to be presented in and when.

IV. APPLICATION OF THE FRAMEWORK

A. APPLICATION OF THE FRAMEWORK

For the purpose of discussing how this framework may be applied, a generic avionics software component is chosen as the reusable software component and the safety-critical system of systems is the U.S. Naval aviation system of systems. It is assumed that there is a desire among stakeholders to reuse this avionics software in future aircraft systems or additional platforms within a system of systems (as evident in Navy's open architecture strategy, which among other things seeks to enable component reuse). The system of systems includes the aircraft, air traffic control, communication systems, naval vessels and bases, and the ground- and air-based systems of all those elements interoperating with Naval aviation to achieve the common mission.

1. Example Process Applying the Framework

Although the framework for software reuse in safety-critical system of systems is process neutral, this example application of the framework will utilize a process for demonstration purposes only. The framework is applied throughout the software life cycle of the reusable software component and is shown in figure 4.

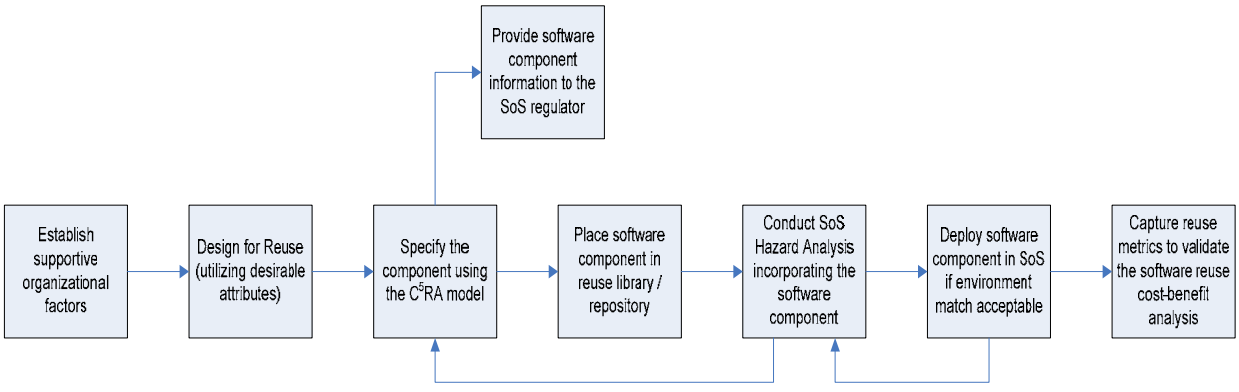

Figure 4. Example Process Showing Application of the Framework

When developing this avionics software component and determining its acceptability for reuse in the environment, the following example approach may be applied:

- Establish organizational factors that are supportive of reuse to ensure that incongruent organizational factors will not undermine a reuse approach. Create the appropriate organizational structure to manage the software component and a software repository staffed with the appropriate people to manage it. Invest the necessary resources in the reuse effort to ensure that the software component is suitable for reuse with significant stakeholder support for the decision to design for reuse. Align organizational policies so that they incentivize reuse and ensure that it is managed effectively.
- Design for reuse through the appropriate investments in quality and the utilization of the desirable software component attributes described in §3C3. Apply the software safety metrics described in §3F2 to indicate progress in designing in safety in their respective order of significance at the applicable stage throughout development to indicate progress or achievement in minimizing the mishap risk. The software safety metrics are applied to ensure that safety is considered throughout the software development process.
- Specify and document the software component using the guidelines in §3D2. Ensure that all information on the component remains with the component including the C^5RA software component specification model utilizing an appropriate stakeholder (including regulator) endorsed ontology and the respective software artifacts for the software component. The specification of the component must facilitate searching by a respective future user (i.e., specification matching). That is, a potential user must be able to decide whether the component meets their requirement and provide more detailed information once a decision has been made to investigate the component further. The C^5RA model should be used as a high level abstract description of the component, coupled with an appropriate stakeholder-endorsed ontology to assist in providing the information required by future potential users and

maintainers of the software as it evolves. All software artifacts for the component should be properly documented in the respective accepted languages and models and provided with the C[5]RA model information.

- The component should be placed in the software repository where that information can be easily accessed, supported by tools, and managed by the appropriate people in accordance with policy.
- Software component information should be made available to the system of systems regulator, to provide early software component descriptions and information that is necessary for the regulator to be actively involved in the certification process. This information is to be provided to the regulator (certification authority) regardless of whether it has been provided to the applicant, integrator, or user in order to achieve certification for operation in the system of systems. The exact timings for provision of this information will be dependent on the particular regulator.
- Perform a system-of-systems hazard analysis that incorporates the reusable software component. This system-of-systems hazard analysis will include the identification, analysis, and treatment of all system of system hazards (i.e., system hazards and emergent hazards). This analysis information will be used as input to the safety case and to update the Analysis element of the C[5]RA model which will subsequently be provided to the regulator as required. Evidence must be obtained and captured that is supportive of an acceptable level of safety risk (that is, the safety-critical system of systems to contain the component is considered "safe enough"[76]).
- Deploy the software component in the system of systems if the system-of-systems hazard analysis (environment match) is supportive of an acceptable level of safety or mishap risk and the regulator accepts. Operational

76 Safe enough may be defined as that point where the benefits of the system outweigh the risks to be determined on a case-by-case basis by the relevant system stakeholders.

information is used to constantly update to system-of-systems hazard analysis to ensure an acceptable level of safety / mishap risk.

- Capture reuse metrics throughout the software life cycle and update the reuse cost-benefit analysis in order to measure the success of the reuse program. This will ensure that reuse performance is measured to enable leveraged success and to populate and update institutional memory.
- When the decision is made to reuse the software component (leveraging off that information in the component specification) within the existing system of systems or in another system of systems, the safety analysis (system-of-systems hazard analysis) is to be repeated with analysis information used to update the component specification and subsequently provided to the regulator.

B. CONCLUSION

To demonstrate how the framework may be applied, a process was chosen and a generic avionics software component used as the reusable software component. This demonstration provided an example and discussion of how the framework may be applied to safety-critical system of systems where software reuse is desired.

V. CONCLUSION

A. KEY FINDINGS AND ACCOMPLISHMENTS

Software reuse is viewed as a means for achieving rapid system development, saving resources and time, and keeping up technologically in an increasingly advancing global environment. Reuse offers many benefits and yet much of the software developed today is still newly developed but not unique. Much of the problem in making software reuse more prevalent is that software was not designed with reuse in mind and that not enough readily available information is shared on a software component. Within a safety-critical system of systems, the demand for information on the software components is even greater and more critical.

At present there is very little guidance in standards on how to best utilize software reuse within an industry or in a system of systems. To best utilize software reuse, we created a framework to enable reuse within a safety-critical system of systems. That framework consists of the enabler made up of organizational factors and three pillars: component attributes (quality), component specification (information capture and search effectiveness), and safety analysis. Congruence between all framework elements is required for software reuse to be a success in a safety-critical system of systems and any one element that is not supportive may lead to failure. This research focused on developing the framework and describing all elements required for effective reuse. In this context more information on a software component is required for reuse due to the complexities and potential system configurations. In essence, a system of systems is system reuse where existing systems are reused in larger system configurations for potentially different missions.

This research described the attributes of reusable software components, including their safety-supportive metrics and how they should be specified, and explored a process for performing a system-of-systems hazard analysis. The software component attributes are based on well accepted software engineering principles and those metrics that support the integration of system safety into the design of the software component. Software-

safety metrics were identified, ranked according to their importance, and described by their relevance in the software life cycle. Component specification expanded on a set of models for describing components and provided an abstract description of the software component for performing a specification match as well as including all relevant information with the component itself. The system-of-systems hazard analysis discussed the environment match of the software component that must be supportive of the component's use and be at an acceptable level of mishap risk once treatment strategies have been enacted.

This research leveraged off other efforts to make software reuse more prevalent, such as the FAA guidance on reusable software components and the U.S. Navy's open architecture strategy. The FAA guidance was thorough on software reuse but focused on collaboration, communication, and product deliverables rather than a systematic approach across the entire software life cycle.

A process then decribed how the framework may be applied to the reuse of a generic avionics software component within an aviation system of systems context.

It is important to realize that it takes a concerted effort and commitment from many people to make software reuse a reality, especially in the context of safety-critical system of systems. Organizational factors must support reuse across stakeholders; the software component must be designed for reuse incorporating those principles and properties that support reuse and designing in of quality; the software component must be specified accordingly to provide the necessary information and to facilitate more effective search (specification matching); and finally the software component must fit the proposed deployment environment. Reuse should not blindly occur without consideration of the environment in which the system will be deployed.

B. FUTURE WORK

One avenue of future work is to refine the specification model with a fully developed ontology with the aim of incorporating further precision into the specification

and building a better shared understanding by stakeholders. This ontology is likely to be domain or system of systems domain specific and may require regulator and stakeholder endorsement for its success.

Additionally, future work could involve developing a case study or the testing of the framework in a system of systems setting with the aim of refining its applicability to the safety-critical system of systems domain.

Further work could also include the development of additional metrics that deal explicitly with reuse components for safety-critical system of systems.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: EXAMPLE SOFTWARE LEVELS OF RIGOR

A. EXAMPLE SOFTWARE LEVEL OF RIGOR (LOR) MATRIX AND REQUIRED LEVEL OF RIGOR SOFTWARE PRODUCTS[77]

Hazard Severity	Software Level Autonomy					
	(I) Software Exercises autonomous control over potentially hazardous hardware systems, subsystems or components without the possibility of intervention to preclude the occurrence of the hazardous event.	(IIa) Software Exercises control over potentially hazardous hardware systems, subsystems or components allowing time for intervention by independent safety systems to mitigate the hazard. However, these systems by themselves are considered not adequate.	(IIb) Software item displays information requiring immediate operator action to mitigate a hazard. Software failures will allow or fail to prevent the hazard's occurrence.	(IIIa) Software item issues commands over potentially hazardous hardware systems, subsystems or components requiring operator action to complete the control function. There are several, redundant, independent safety measures for each hazardous event.	(IIIb) Software generates information of a safety critical nature used to make safety critical decisions. There are several, redundant, independent safety measures for each hazardous event.	(IV) Software does not control safety critical hardware systems, subsystems or components and does not provide safety critical information.
Catastrophic (I)	LOR3	LOR3	LOR3	LOR2	LOR2	N/A
Critical (II)	LOR3	LOR2	LOR2	LOR2	LOR2	N/A
Marginal (III)	LOR2	LOR2	LOR2	LOR1	LOR1	N/A
Negligible (IV)	LOR1	LOR1	LOR1	LOR1	LOR1	N/A

Table 4. Example Software Level of Rigor Matrix

77 V. Basili., K. Dangle., L. Esker., F. Marotta., Gaining Early Insight into Software Safety: Measures of Potential Problems and Risks, in Proc of the Systems & Software Technology Conference, June 2007, p. 34.

Level Of Rigor (LOR)	Software Development Products
LOR 3 – Highest	**Code Walkthroughs**
	Condition / Decision structural test with safety mitigation records
	All products in lower levels
LOR 2	**Design analysis with updates to requirements hazard analysis products**
	Functional hazard analysis
	Functional testing
	Stress and stability testing
	All products in lower levels
LOR 1 – Lowest	**Software Safety Requirements**
	Hazard Mitigation Traceability Matrix
	Functional or system hazard analysis
	Hazard control Records
	Computer Program Change Requests
	System or Functional Testing

Table 5. Example Required Level of Rigor Software Products

B. SOFTWARE RISK ASSESSMENT MATRIX AND SOFTWARE HAZARD RISK INDEX[78]

Hazard Severity Category Software Control Category	Catastrophic	Critical	Marginal	Negligible
I	1	1	3	5
II	1	2	4	5
III	2	3	5	5
IV	3	4	5	5

Table 6. Software Risk Assessment Matrix

Hazard Risk Index	Criteria
1	High Risk: Significant Analysis and Testing resources required.
2	Medium Risk: Requirements and design Analysis and in-depth testing required.
3-4	Moderate Risk: High level analysis and testing acceptable with Managing Activity approval.
5	Low Risk: Acceptable.

Table 7. Software Hazard Risk Index

78 J.B. Michael., SW4582 Weapon System Software Safety Lecture Notes, Naval Postgraduate School, 2006, Module 2, Part 3, p. 19-20.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Advisory Circular AC 20-148, Reusable Software Components, US Department of Transportation, Federal Aviation Administration, December 7, 2004.

Anthony, R. Software Firms Reap Benefits of Code Reuse. The Wall Street Journal, December 3, 2007. www.livemint.com.

AS/NZS 4360:2004. Australian / New Zealand Standard. Risk Management.

Atchison, B., Wabenhorst, A. A Survey of International Safety Standards, Technical Report No. 99-30, Software Verification research centre, School of Information Technology, University of Queensland, November 1999.

Basili, V., Dangle, K., Esker, L., Marotta, F. Gaining Early Insight into Software Safety: Measures of Potential Problems and Risks, in Proc of the Systems & Software Technology Conference, June 2007, pp. 1–40.

Basili, V., Weiss. D. A Methodology For Collecting Valid Software Engineering Data, IEEE Transactions on Software Engineering, Volume 10, Number 6, October 1984, pp. 728–738.

Braude, E., Software Design: From Programming to Architecture, Hoboken, NJ.: John Wiley & Sons, 2004.

Brown, M.L., Safety Critical Software Development, in Proc. of the Center for National Software Studies Workshop on Trustworthy Software, May 2004, pp. 11–12.

Caffall, D.S., Michael, J.B. Architectural Framework for a System-of-Systems, in Proc of the IEEE International Conference on Systems, Man and Cybernetics, Volume 2, 10-12 Oct 2005, pp. 1876–1881.

Carvallo, J.P., Franch, X., Quer, C. Determining Criteria for Selecting Software Components: Lessons Learned, in IEEE Software, Volume 24, Issue 3, May–June 2007, pp. 84-94.

Department of Defense Directive 5000.1, The Defense Acquisition System, 12 May 2003.

Department of Defense, Standard Practice for System Safety, MIL-STD-882D, 10 February 2000.

Dependability.org, IFIP WG-10.4, November, 2006, http://www.dependability.org.

Drsinsky, D., Michael, J.B., Shing, M. The Three Dimensions of Formal Validation and Verification of Reactive System Behaviors, Naval Postgraduate School Technical Report NPS-CS-07-008, Monterey, CA, August 2007.

Dunn, W.R. Practical Design of Safety Critical Computer Systems. Solvang, CA.: Reliability Press, 2002.

Frakes, W., & Terry, C. Software Reuse: Metrics and Models. ACM Computing Surveys, Volume 28, Number 2, June 1996, pp. 415–435.

IEEE standard for Software Quality Assurance Plans, 730-1998.

Institute of Electrical and Electronics Engineers, Standard 1028-1997, IEEE Standard for Software Reviews, 1997.

Joyce, E.J. Reusable Software: Passage to Productivity. Datamation, Volume 34, Number 18, Spetember 15, 1988, pp. 97–102.

Kilmann, R.H., Saxton, M.J., Serpa, R., Gaining Control of the Corporate Culture, Jossey Bass Business and management Series, San Francisco, CA.: 1985.

Larman, C., Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition, Upper Saddle River, NJ.: Pearson Education, 2005.

Leveson, N. Safeware: System Safety and Computers. Addison Wesley, Boston, 1995.

Lim, W.C. Managing Software Reuse, A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components, Upper Saddle River, NJ.: Prentice Hall PTR, 1998.

Meyer, M. Object-Oriented Software Construction, Second Edition, Upper Saddle River, NJ.: Prentice Hall PTR, 1997.

Michael, J.B. SW4582 Weapon System Software Safety Lecture Notes, Naval Postgraduate School, 2006.

National Aeronautics and Space Administration. System Safety Handbook (DHB-S-001). Dryden Research Flight Center, Edwards, CA, 1999.

NATO Software Engineering Conference, NATO Science Committee, Garmisch, Germany, 7-11 October 1968.

Overview of Software Safety, NASA Software Safety, May 2007, http://sw-assurance.gsfc.nasa.gov/disciplines/safety/index.php.

Pressman, R.S., Software Engineering A Practitioner's Approach, Sixth Edition, New York, NY.: McGraw-Hill, 2005.

Radio Technical Commission for Aeronautics, DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification, Washington D.C., 1 December 1992.

Ramachandran, M. Software Reuse Guidelines. ACM SIGSOFT Software Engineering Notes, 30(3), May 2005, pp. 1–8.

Redmond, P. A System of Systems Interface Hazard Analysis Technique, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 2007.

Riehle, R. Software Design Metrics: Designing Software Components to Tolerances, ACM SIGSOFT Software Engineering Notes, Volume 32, Issue 4, July, 2007, pp. 1–6.

Riehle, R. Software Engineering Practice, Failure-driven Software Safety, in ACM SIGSOFT Software Engineering Notes, Volume 32, Number 5, September 2007, pp. 1–4.

Singh, R. A Systematic Approach to Software Safety, in Proc of the sixth Asia Pacific Software Engineering Conference (APSEC '99), 1999, pp. 420–423.

Smith, D.J. Reliability Maintainability and Risk, practical Methods for Engineers, Seventh Edition. Oxford, England.: Elsevier Butterworth-Heinemann, 2005.

Software Engineering Institute, Defence Materiel Organisation Australian Department of Defence. +SAFE, V1.2 A Safety Extension to CMMI-DEV, V1.2, Technical Note CMU/SEI-2007-TN-006, March 2007.

Storey, N., Safety-Critical Computer Systems, New York, NY.: Addison-Wesley, 1996.

Tracz, W. The 3 Cons of Software Reuse, in Proc of the Third Annual Workshop on Software Reuse, July, Syracuse, NY, 1990.

Wbenhorst, A., Atchison, B. A Survey of International Safety Standards. Technical Report No. 99-30, Software Verification Research Centre, School of Information Technology, University of Queensland, November 1999.

Weide, B.W et al., Reusable Software Components, Advances in Computers, Yovits, M. C (ed.), Vol. 33, Academic Press, 1991, pp. 1–65.

Wlad, J. Software Reuse In Safety critical Airborne Systems, in Proc of 25th Digital Avionics Systems Conference, Oct 15, 2006, pp. 6C5-1 – 6C5-8.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Library
   Australian Defence Force Academy
   Campbell, Australian Capital Territory, Australia

4. Colonel Anthony McWatters
   Australian Army
   Brisbane Airport, QLD, Australia

5. Professor Bret Michael
   Naval Postgraduate School
   Monterey, California

6. Professor Man-Tak Shing
   Naval Postgraduate School
   Monterey, California

7. Mr. John Harauz
   Jonic Systems Engineering, Inc.
   Willowdale, Ontario, Canada

8. Mr. Archibald McKinlay
   Naval Ordnance Safety and Security Activity
   Indian head, Maryland

9. Mr. Michael Brown
   EG&G
   Mariposa, California

10. Mr. Nicholas Guertin
    PEO Integrated Warfare Systems
    Washington, D.C.